Running Probabilistic Programs Backwards

Neil Toronto * Jay McCarthy † David Van Horn *

* University of Maryland † Vassar College

ESOP 2015

2015/04/14

Roadmap

- Probabilistic inference, and why it's hard

Roadmap

- Probabilistic inference, and why it's hard
- Limitations of current probabilistic programming languages (PPLs)

Roadmap

- Probabilistic inference, and why it's hard
- Limitations of current probabilistic programming languages (PPLs)
- Contributions

Roadmap

- Probabilistic inference, and why it's hard
- Limitations of current probabilistic programming languages (PPLs)
- Contributions
  - Uncomputable, compositional ways to not limit language

Roadmap

- Probabilistic inference, and why it's hard
- Limitations of current probabilistic programming languages (PPLs)
- Contributions
  - Uncomputable, compositional ways to not limit language
  - Computable, compositional ways to not limit language

Programming Coin Flips

```
(let ([x  (flip 0.5)])
  x)
```


Programming Coin Flips

```
(let ([x  (flip 0.5)])
  x)
```

0.5

Programming Coin Flips

```
(let ([x  (flip 0.5)])
  x)
```

0.5

0.5

Programming Coin Flips

```
(let ([x  (flip 0.5)]
      [y  (flip 0.5)])
  (cons x y))
```

0.5

0.5 ⟨ 

, 

⟩

0.5 ⟨ 

, 

⟩

Programming Coin Flips

```
(let ([x  (flip 0.5)]
      [y  (flip 0.5)])
  (cons x y))
```

Programming Coin Flips

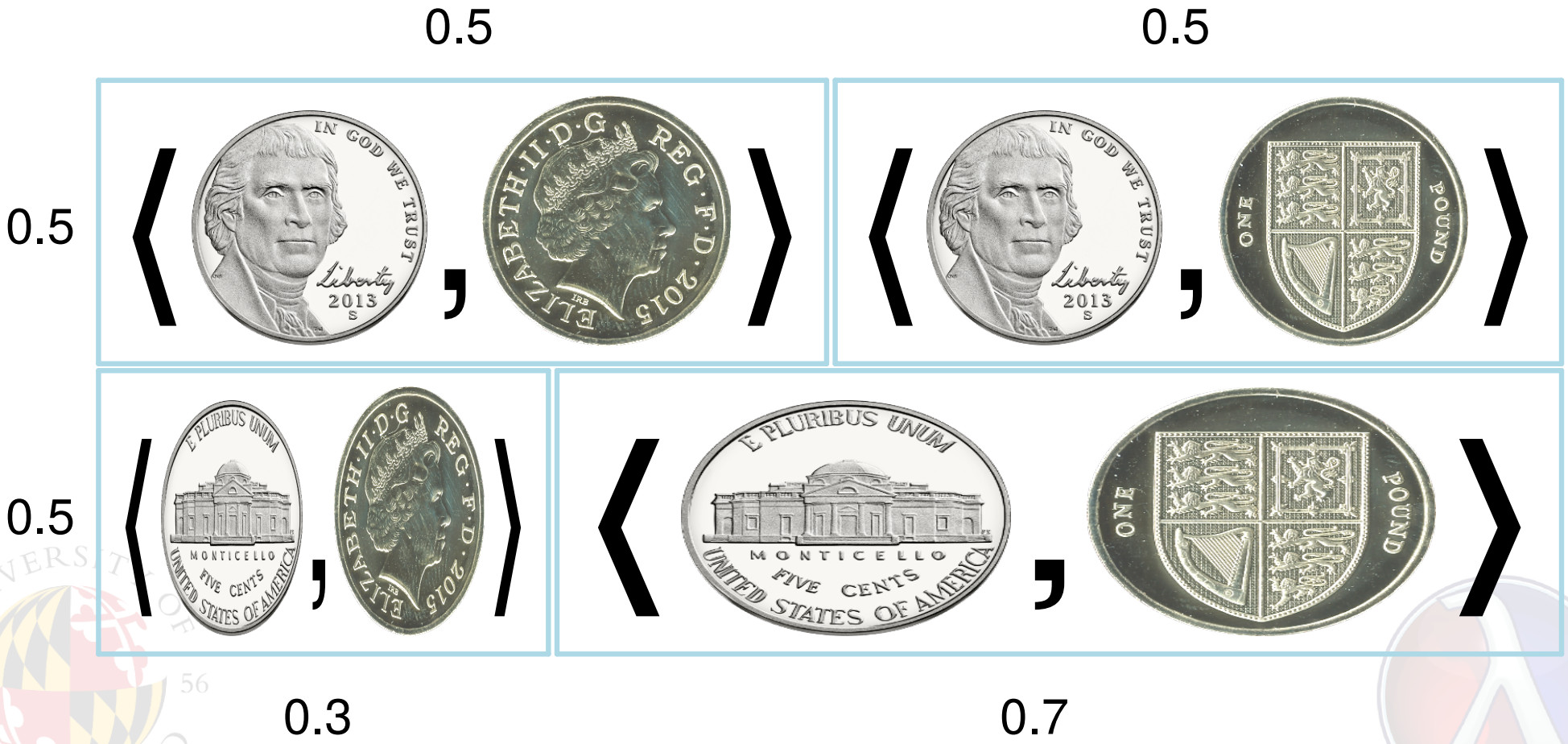
```
(let ([x  (flip 0.5)]
      [y  (flip 0.5)])
  (cons x y))
```

Programming Coin Flips

```
(let* ([x  (flip 0.5)]
       [y  (flip (if (equal? x heads) 0.5 0.3))])
  (cons x y))
```

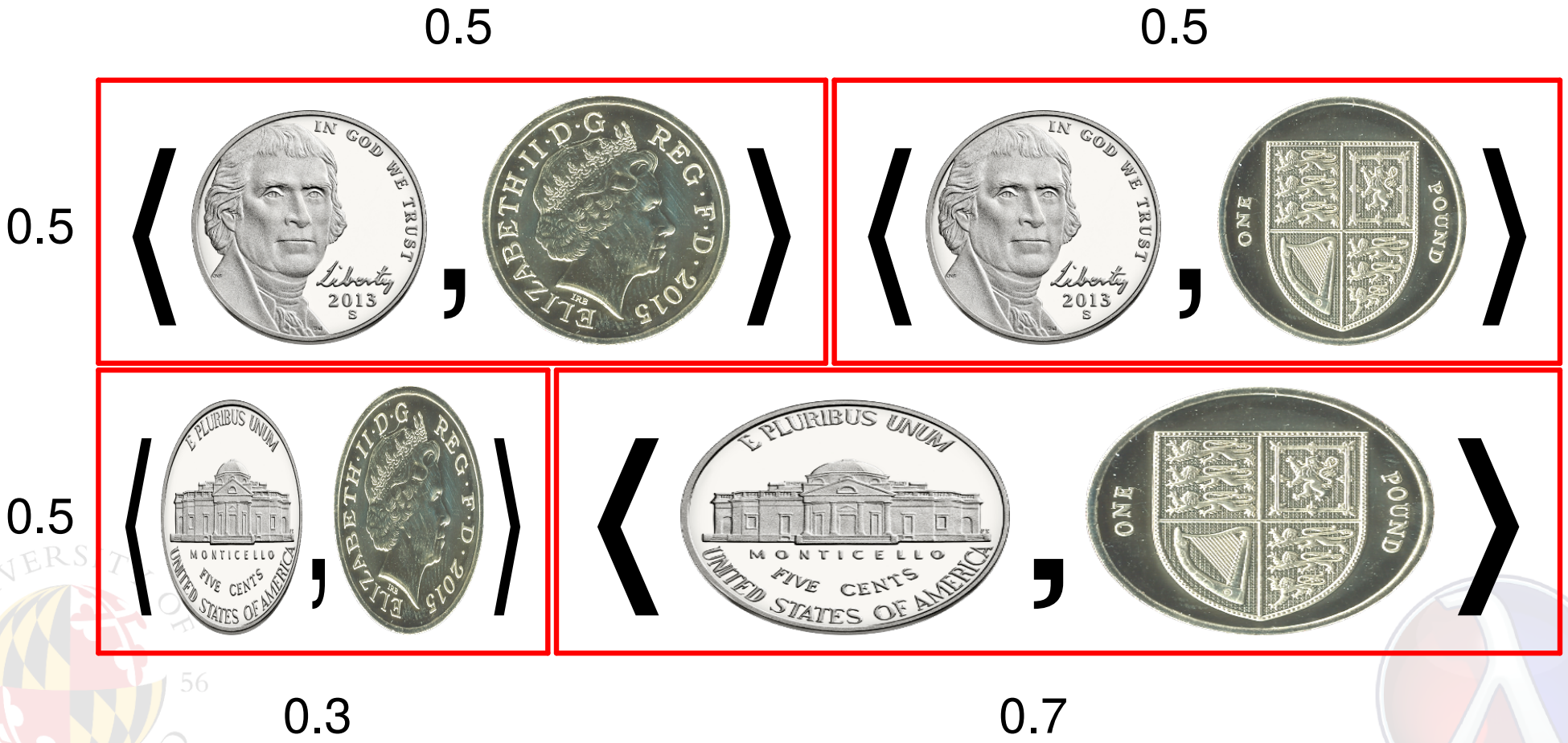
Programming Coin Flips

```
(let* ([x  (flip 0.5)]
       [y  (flip (if (equal? x heads) 0.5 0.3))])
  (cons x y))
```



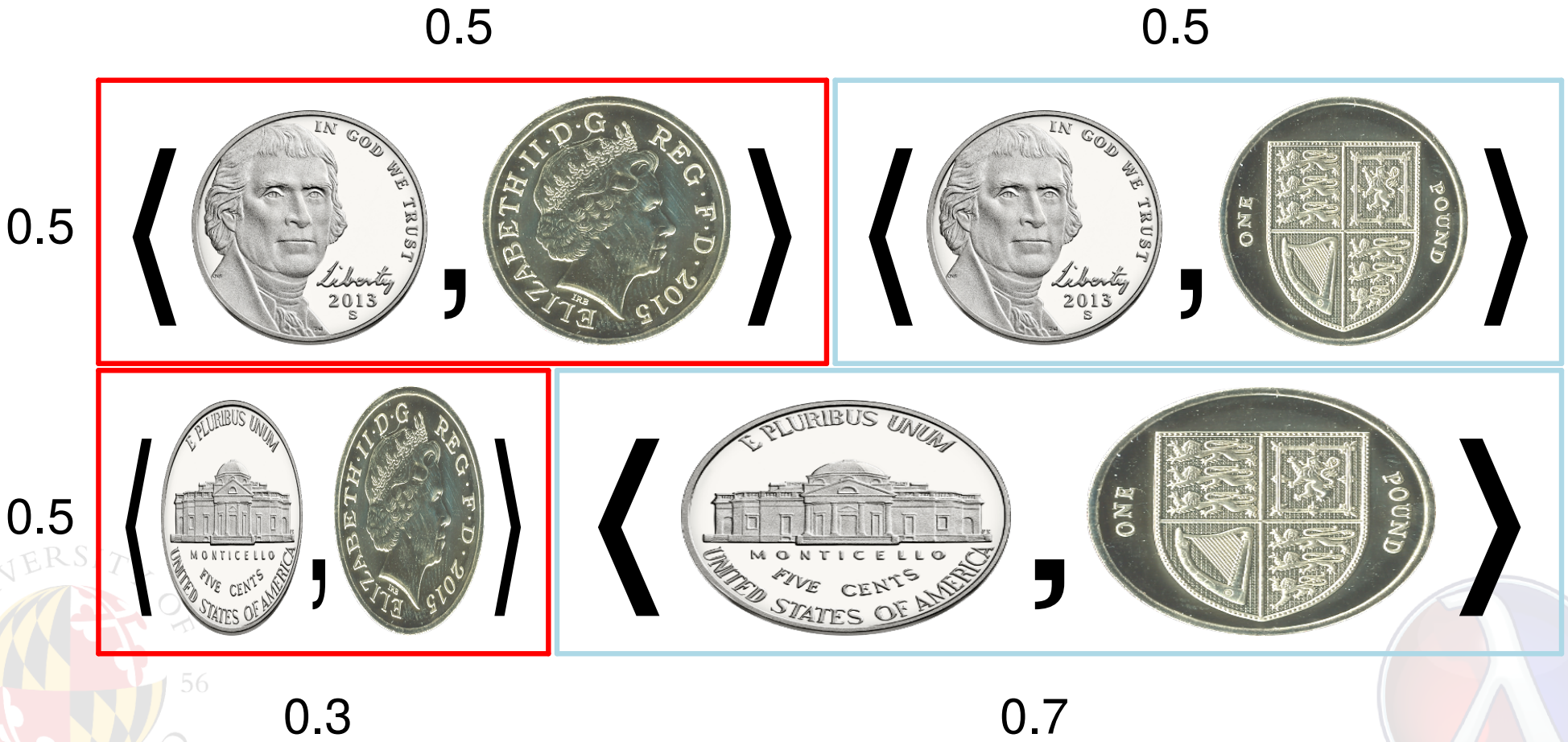
Programming Coin Flips

$$\Pr[\texttt{true}] = 0.5 \cdot 0.5 + 0.5 \cdot 0.5 + 0.5 \cdot 0.3 + 0.5 \cdot 0.7 = 1$$



Programming Coin Flips

$$\Pr[\mathrm{y} = \mathrm{heads}] = 0.5 \cdot 0.5 + 0.5 \cdot 0.3 = 0.4$$



Programming Coin Flips

$$\Pr[\mathrm{x} = \mathrm{heads} \mid \mathrm{y} = \mathrm{heads}]$$
$$= \Pr[\langle \mathrm{x}, \mathrm{y} \rangle = \langle \mathrm{heads}, \mathrm{heads} \rangle] / \Pr[\mathrm{y} = \mathrm{heads}]$$
$$= 0.25/0.4 = 0.625$$

0.5

0.5

0.5

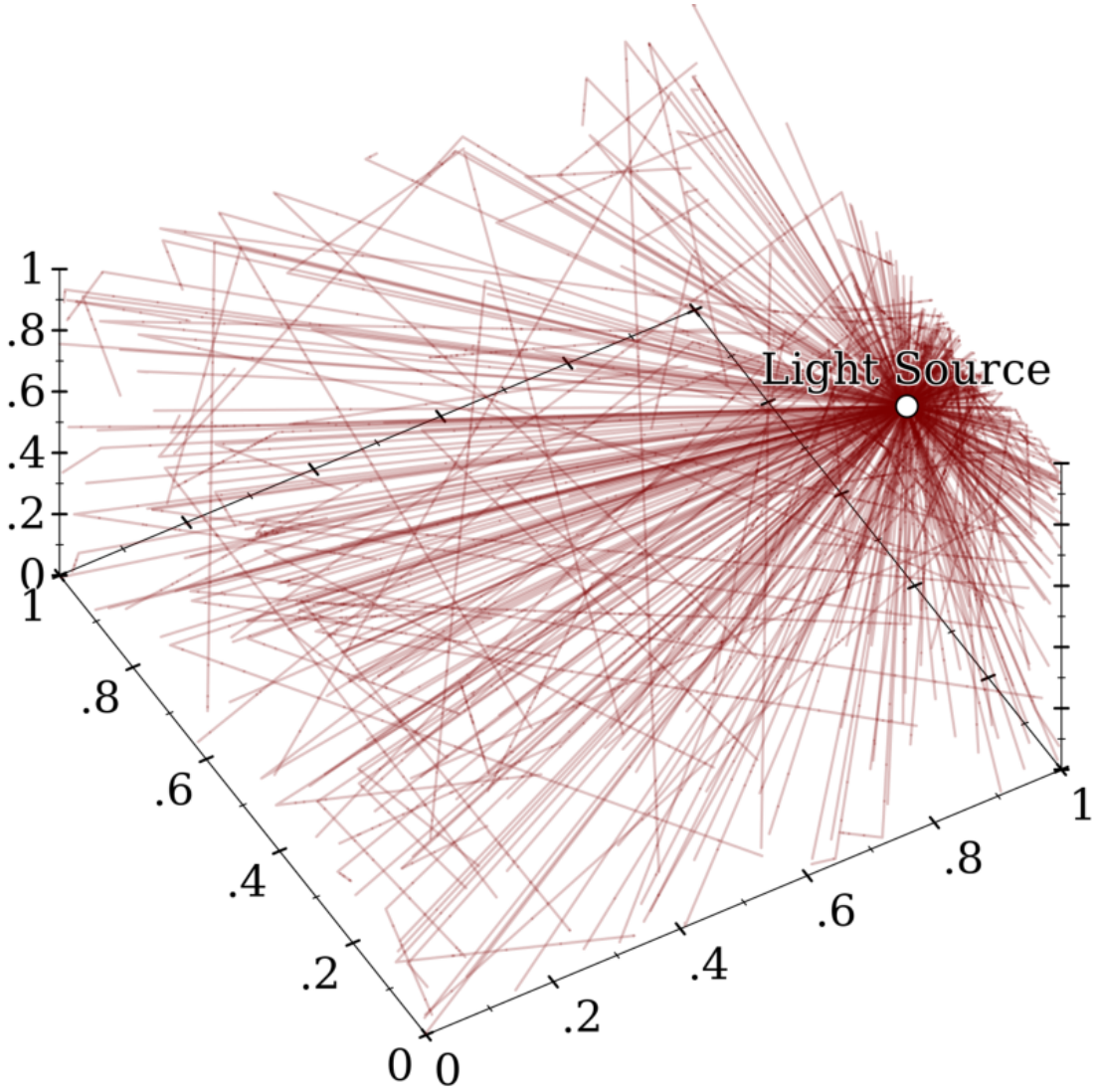
0.3

Stochastic Ray Tracing

sto·cha·stic /stō-ˈkas-tik/ *adj.* fancy word for "randomized"

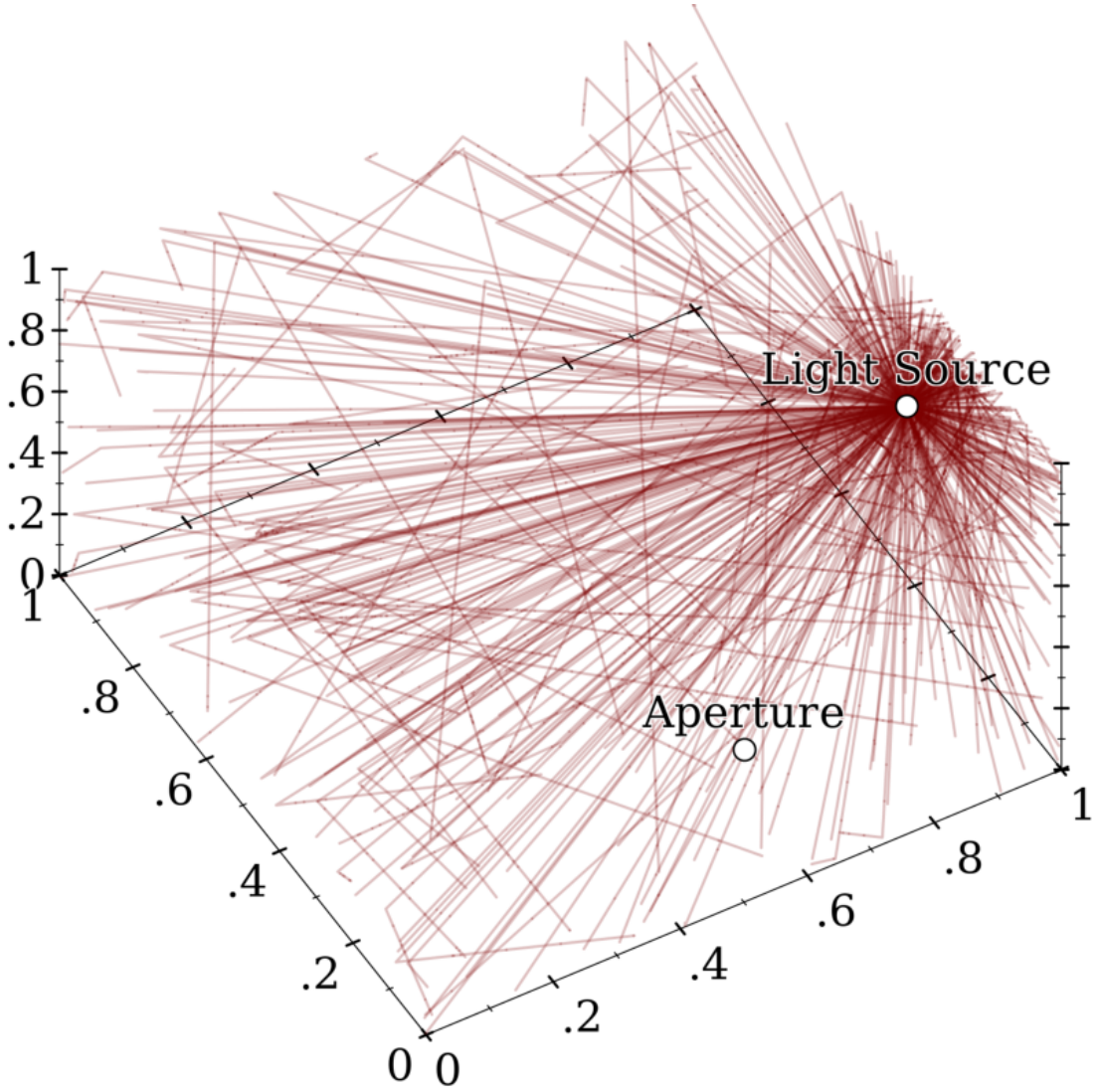
Stochastic Ray Tracing

sto·cha·stic /stō-ˈkas-tik/ *adj.* fancy word for "randomized"



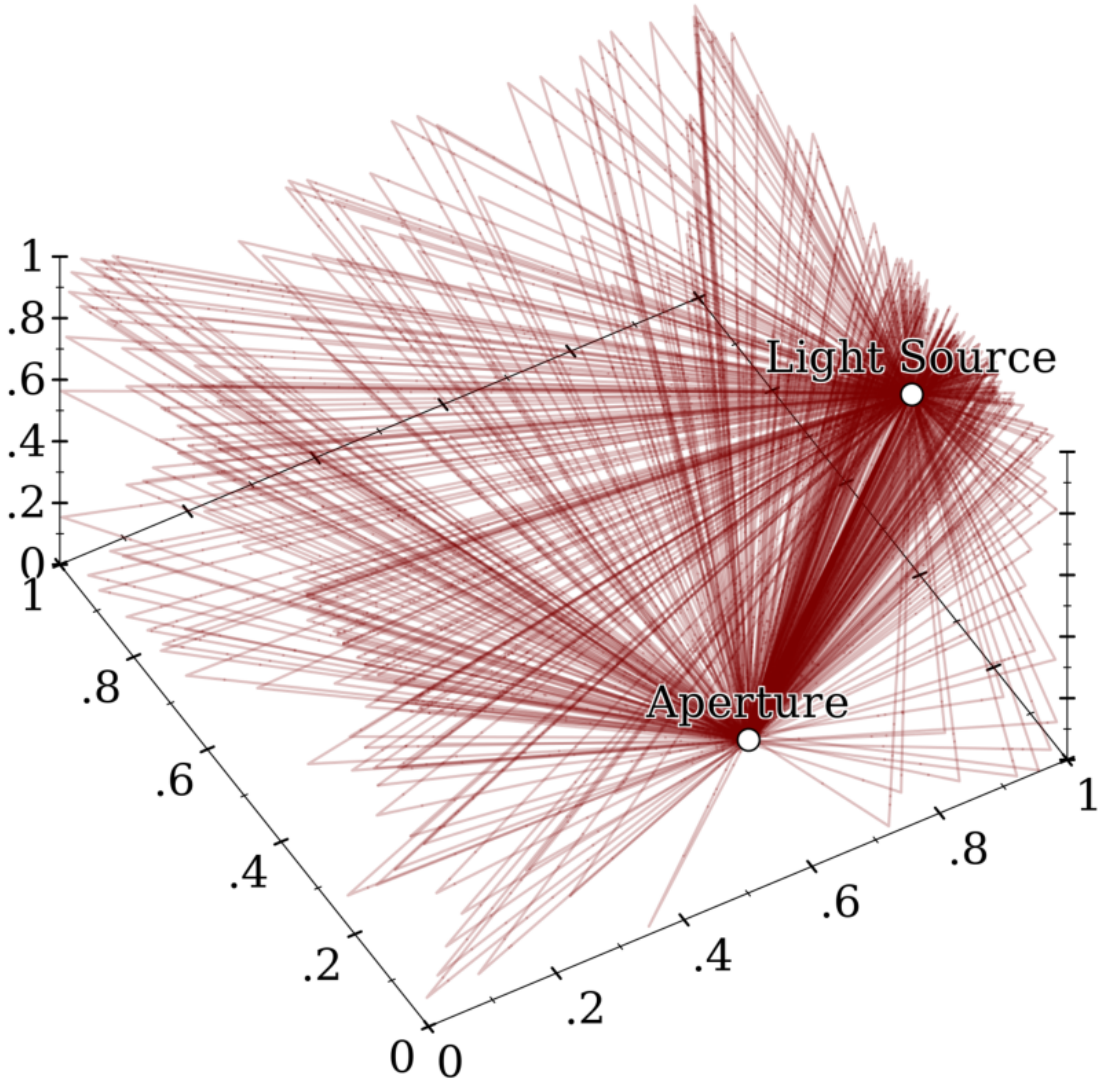
Stochastic Ray Tracing

ap·er·ture /'ap-ə(r)-chər/ *n.* fancy word for "opening"



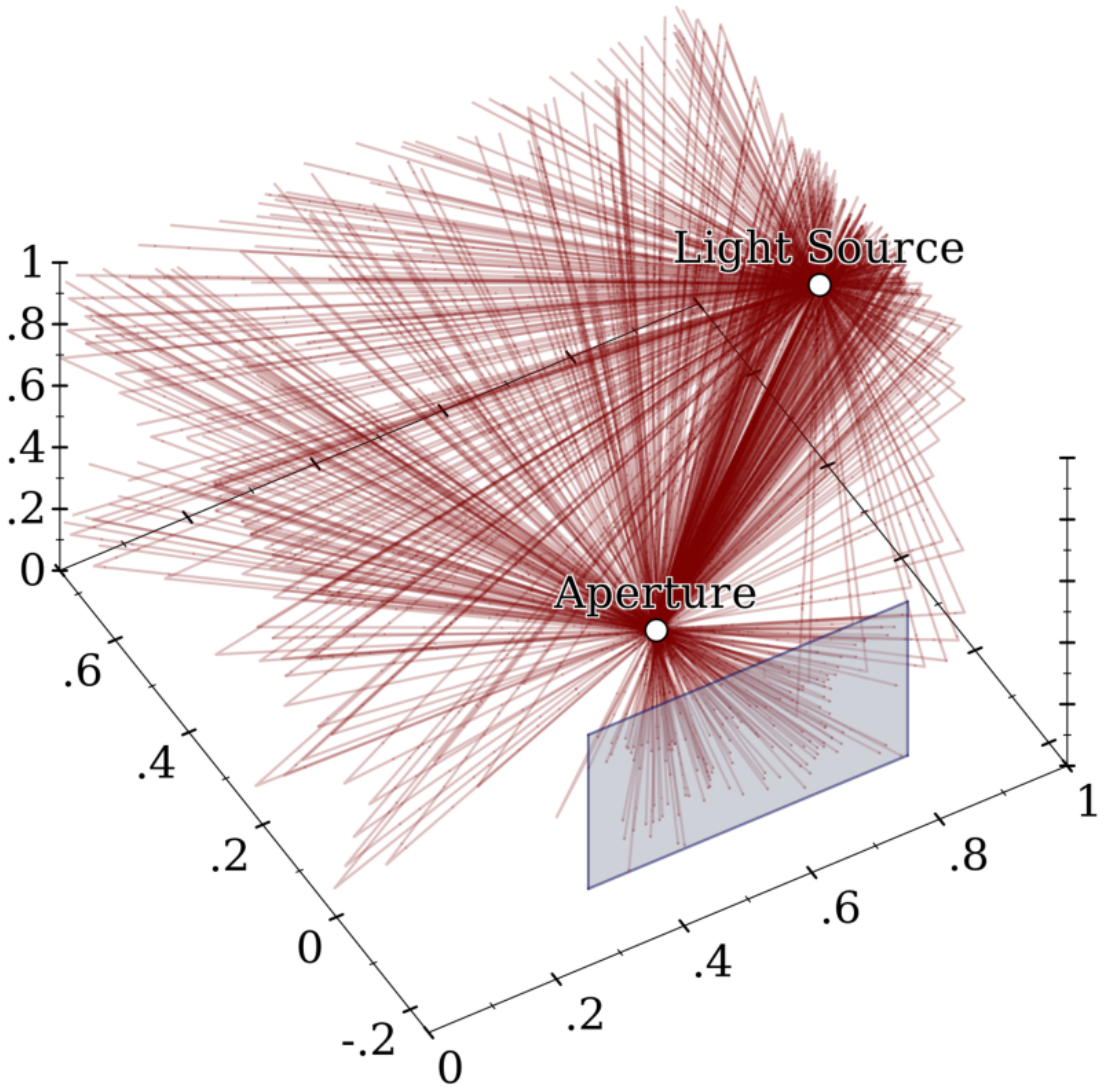
Stochastic Ray Tracing

ap·er·ture /'ap-ə(r)-chər/ *n.* fancy word for "opening"



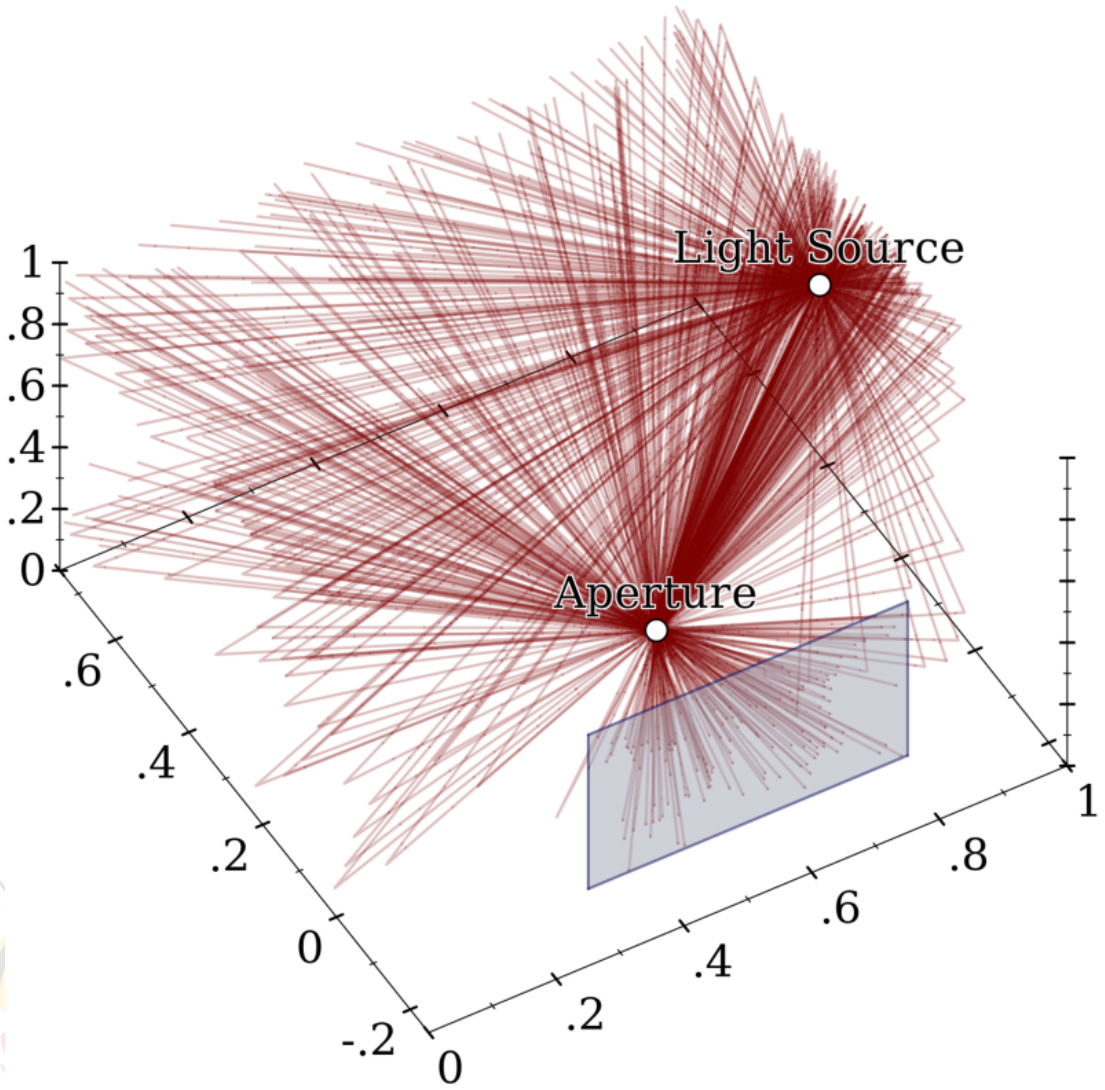
Stochastic Ray Tracing

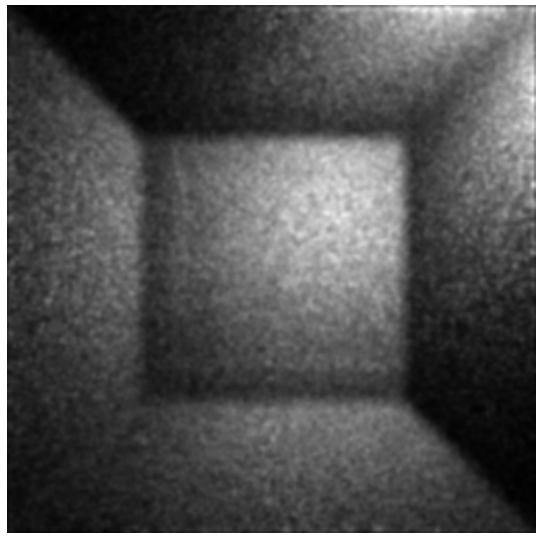
Simulate projecting rays onto a sensor...



Stochastic Ray Tracing

... and collect them to form an image





Programming Stochastic Ray Tracing

- Normally thousands of lines of code

Programming Stochastic Ray Tracing

- Normally thousands of lines of code
- Bears little resemblance to the physical process

Programming Stochastic Ray Tracing

- Normally thousands of lines of code
- Bears little resemblance to the physical process
- In DrBayes, it's simple physics simulation:

```
(define/drbayes (ray-plane-intersect p0 v n d)
  (let ([denom  (- (dot v n))])
    (if (> denom 0)
        (let ([t  (/ (+ d (dot p0 n)) denom)])
          (if (> t 0)
              (collision t (vec+ p0 (vec* v t)) n)
              #f))
        #f)))
```


Programming Stochastic Ray Tracing

- Normally thousands of lines of code
- Bears little resemblance to the physical process
- In DrBayes, it's simple physics simulation:

```
(define/drbayes (ray-plane-intersect p0 v n d)
  (let ([denom  (- (dot v n))])
    (if (> denom 0)
        (let ([t  (/ (+ d (dot p0 n)) denom)])
          (if (> t 0)
              (collision t (vec+ p0 (vec* v t)) n)
              #f))
        #f)))
```

- Other PPLs really aren't up to this yet

Programming Stochastic Ray Tracing

- Normally thousands of lines of code
- Bears little resemblance to the physical process
- In DrBayes, it's simple physics simulation:

```
(define/drbayes (ray-plane-intersect p0 v n d)
  (let ([denom  (- (dot v n))])
    (if (> denom 0)
        (let ([t  (/ (+ d (dot p0 n)) denom)])
          (if (> t 0)
              (collision t (vec+ p0 (vec* v t)) n)
              #f))
        #f)))
```

- Other PPLs really aren't up to this yet
- The issue is one of theory, not engineering effort

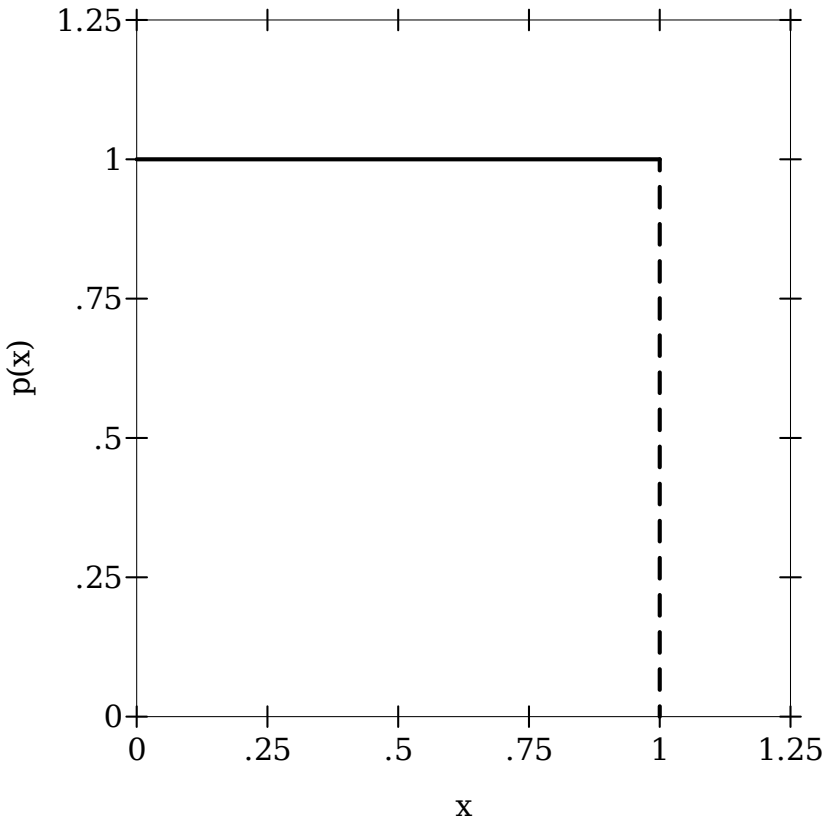
Simpler Example

- Assume `(random)` returns a value uniformly in $[0, 1]$

Simpler Example

- Assume **`(random)`** returns a value uniformly in $[0, 1]$

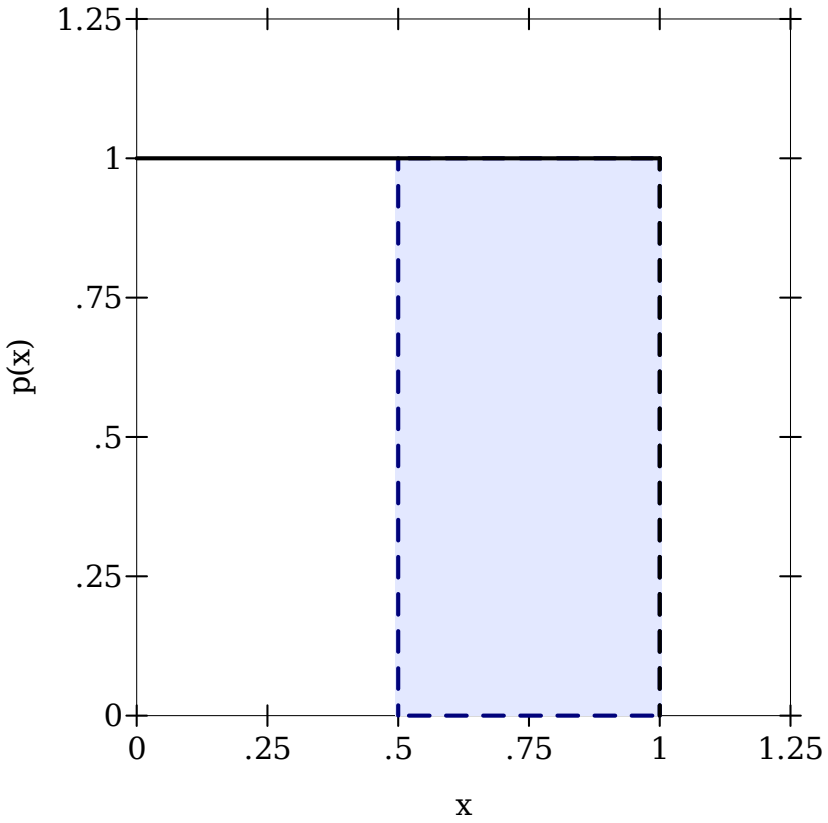
Density function p for value of **`(random)`**:



Simpler Example

- Assume **(random)** returns a value uniformly in $[0, 1]$

Density function p for value of **(random)**:

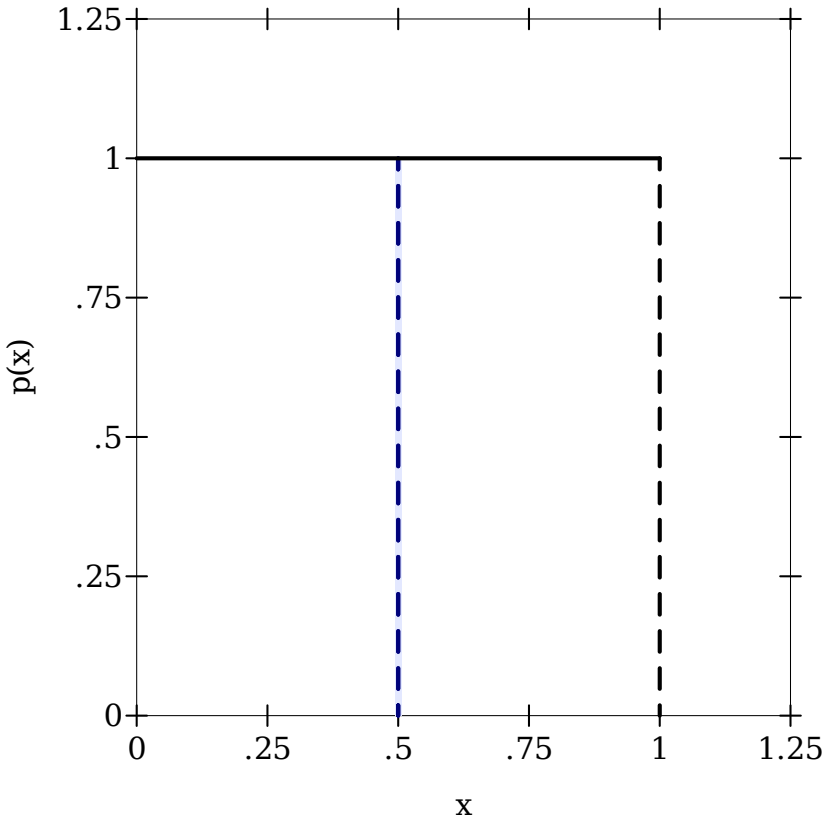


$$\begin{aligned}\Pr[(\texttt{random}) \in [0.5, 1]] &= \int_{0.5}^{1} p(x)\, dx \\ &= 1 - 0.5 \\ &= 0.5\end{aligned}$$

Simpler Example

- Assume **(random)** returns a value uniformly in $[0, 1]$

Density function p for value of **(random)**:

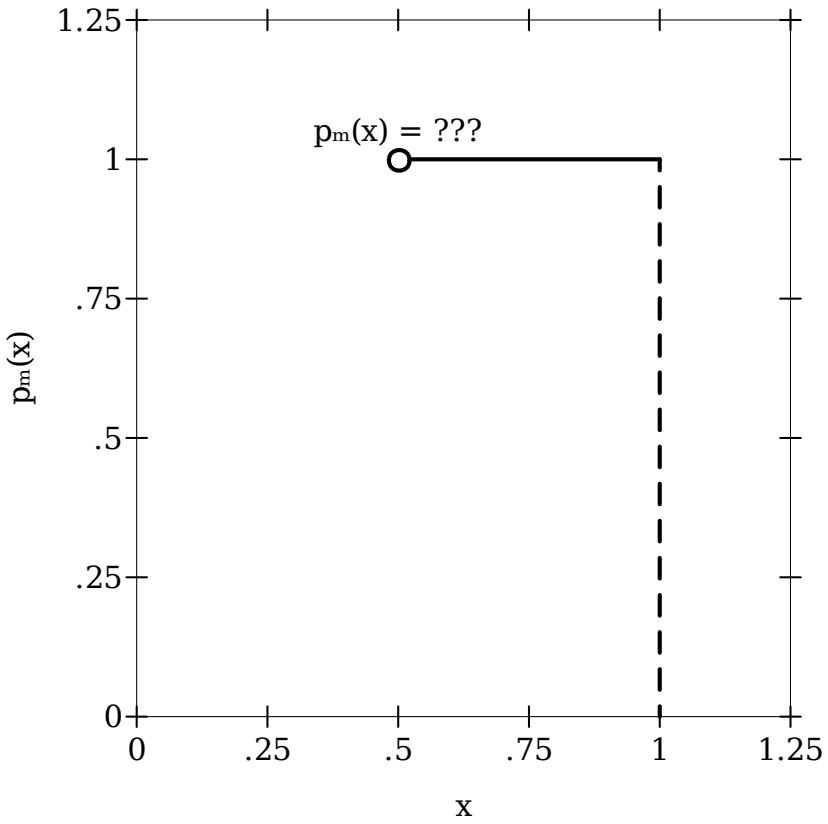


$$\begin{aligned}
&\Pr[(\texttt{random}) \in [0.5, 0.5]] \\
&\quad = \int_{0.5}^{0.5} p(x)\, dx \\
&\quad = 0.5 - 0.5 \\
&\quad = 0
\end{aligned}$$

Simpler Example

- Assume **`(random)`** returns a value uniformly in $[0, 1]$

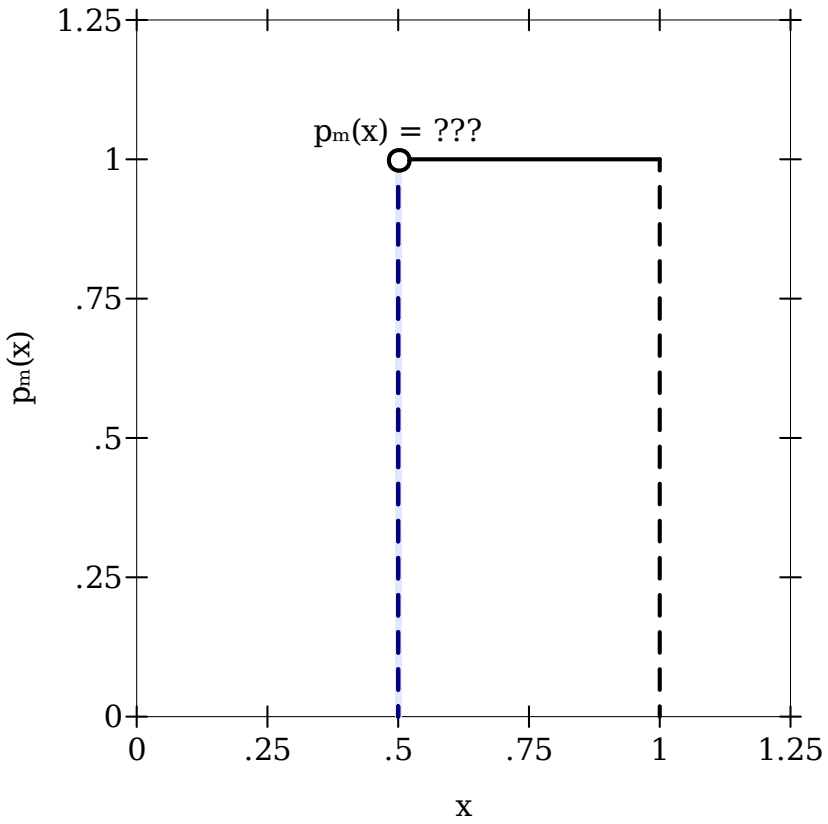
Density function p_m for value of **`(max 0.5 (random))`**:



Simpler Example

- Assume **(random)** returns a value uniformly in $[0, 1]$

Density function p_m for value of **(max 0.5 (random))**:

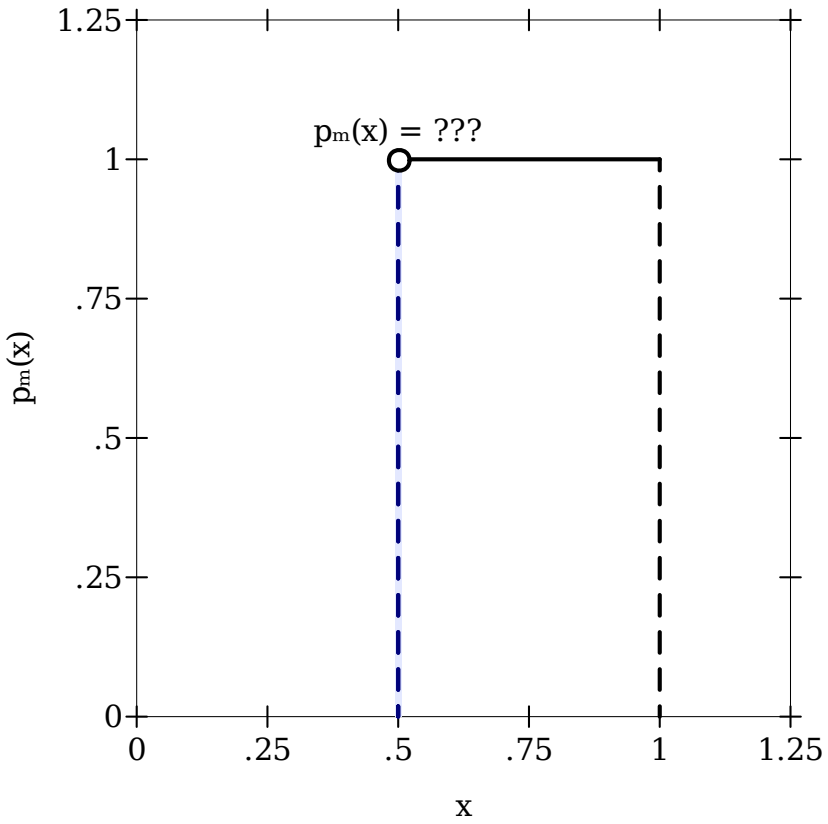


$$\Pr[(\texttt{max } 0.5 \ (\texttt{random})) \in [0.5, 0.5]] = 0.5$$

Simpler Example

- Assume **(random)** returns a value uniformly in $[0, 1]$

Density function p_m for value of **(max 0.5 (random))**:

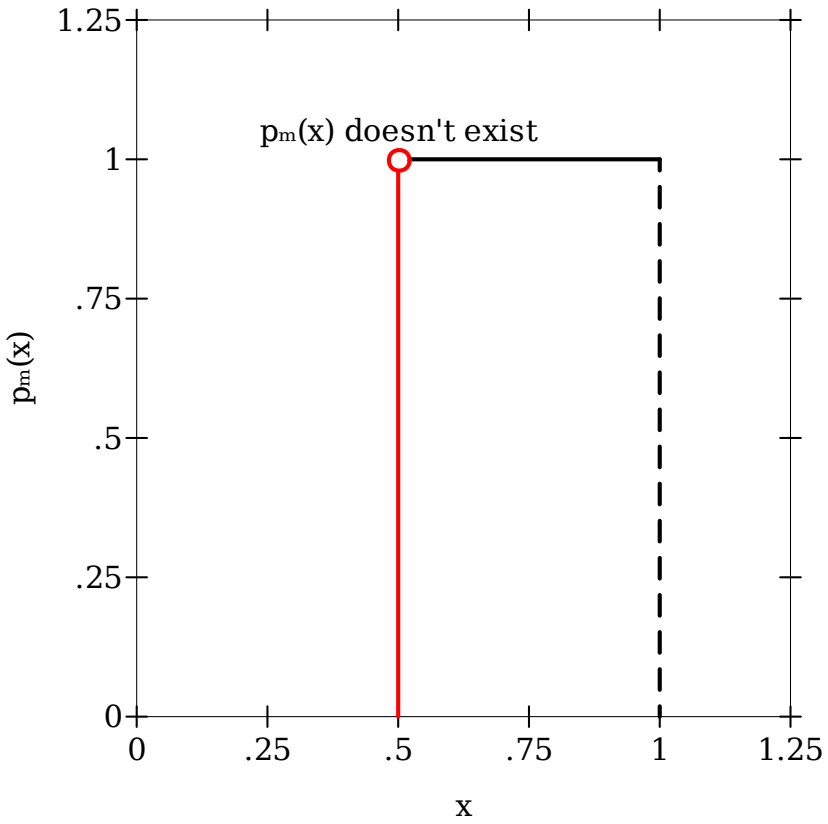


$$\begin{aligned}\Pr[(\texttt{max } 0.5 \ (\texttt{random})) \in [0.5, 0.5]] \\ &= 0.5 \\ &= \int_{0.5}^{0.5} p_m(x)\, dx \\ &= 0\end{aligned}$$

Simpler Example

- Assume **`(random)`** returns a value uniformly in $[0, 1]$

Density function p_m for value of **`(max 0.5 (random))`**:



$$\begin{aligned}\Pr[(\texttt{max } 0.5 \ (\texttt{random})) \in [0.5, 0.5]] \\ &= 0.5 \\ &= \int_{0.5}^{0.5} p_m(x)\, dx \\ &= 0\end{aligned}$$

What Can't Densities Model?

What Can't Densities Model?

- Results of discontinuous functions (bounded measuring devices)

```
(let ([temperature  (normal 99 1)])
  (min 100 temperature))
```


What Can't Densities Model?

- Results of discontinuous functions (bounded measuring devices)

```
(let ([temperature  (normal 99 1)])
  (min 100 temperature))
```

- Variable-dimensional things (union types)

```
(if test? none (just x))
```


What Can't Densities Model?

- Results of discontinuous functions (bounded measuring devices)

```
(let ([temperature  (normal 99 1)])
  (min 100 temperature))
```

- Variable-dimensional things (union types)

```
(if test? none (just x))
```

- Infinite-dimensional things (recursion)

What Can't Densities Model?

- Results of discontinuous functions (bounded measuring devices)

```
(let ([temperature  (normal 99 1)])
  (min 100 temperature))
```

- Variable-dimensional things (union types)

```
(if test? none (just x))
```

- Infinite-dimensional things (recursion)
- In general: the distributions of program values

Probability Measures

- Like already-integrated densities, but a primitive concept

Probability Measures

- Like already-integrated densities, but a primitive concept
- Measure of **(random)** is $P : \mathcal{P}\ [0,1] \rightharpoonup [0,1]$, defined by

$$P\ [a,b] \ = \ \int_a^b p(x)\,dx \ = \ b - a$$

Probability Measures

- Like already-integrated densities, but a primitive concept
- Measure of **(random)** is $P : \mathcal{P}\ [0,1] \rightharpoonup [0,1]$, defined by

$$P\ [a,b] \ = \ b - a$$

Probability Measures

- Like already-integrated densities, but a primitive concept
- Measure of `(random)` is $P : \mathcal{P}\ [0,1] \rightharpoonup [0,1]$, defined by

$$P\ [a,b] \ = \ b - a$$

- Measure of `(max 0.5 (random))` defined by

$$P_m\ [a,b] \ = \ \max(0.5, b) - \max(0.5, a) + \begin{cases} 0.5 & \text{if } a \leq 0.5 \leq b \\ 0 & \text{otherwise} \end{cases}$$

Probability Measures

- Like already-integrated densities, but a primitive concept
- Measure of **(random)** is $P : \mathcal{P}\ [0,1] \rightharpoonup [0,1]$, defined by

$$P\ [a,b] \ = \ b - a$$

- Measure of **(max 0.5 (random))** defined by

$$P_m\ [a,b] \ = \ \max(0.5,b) - \max(0.5,a) + \begin{cases} 0.5 & \text{if } a \leq 0.5 \leq b \\ 0 & \text{otherwise} \end{cases}$$

This term assigns $[0.5, 0.5]$ probability 0.5

Probability Measures

- Like already-integrated densities, but a primitive concept
- Measure of **(random)** is $P : \mathcal{P}\ [0,1] \rightharpoonup [0,1]$, defined by

$$P\ [a,b] \ = \ b - a$$

- Measure of **(max 0.5 (random))** defined by

$$P_m\ [a,b] \ = \ \max(0.5,b) - \max(0.5,a) + \begin{cases} 0.5 & \text{if } a \leq 0.5 \leq b \\ 0 & \text{otherwise} \end{cases}$$

This term assigns $[0.5, 0.5]$ probability 0.5

- Need a way to *derive* measures from code

Probability Measures Via Preimages

- Interpret `(max 0.5 (random))` as $f : [0, 1] \to \mathbb{R}$, defined

$$f\ r \ = \ \max(0.5, r)$$

Probability Measures Via Preimages

- Interpret `(max 0.5 (random))` as $f : [0, 1] \to \mathbb{R}$, defined

$$f\ r\ =\ \max(0.5, r)$$

- *Derive* measure of `(max 0.5 (random))` as

$$P_m\ B = P\ (f^{-1}\ B)$$

Probability Measures Via Preimages

- Interpret `(max 0.5 (random))` as $f : [0,1] \to \mathbb{R}$, defined

$$f\ r\ =\ \max(0.5, r)$$

- *Derive* measure of `(max 0.5 (random))` as

$$P_m\ B = P\ (f^{-1}\ B)$$

where $f^{-1}\ B\ =\ \{r \in [0,1] \mid f\ r \in B\}$

Probability Measures Via Preimages

- Interpret `(max 0.5 (random))` as $f : [0, 1] \to \mathbb{R}$, defined

$$f\ r\ =\ \max(0.5, r)$$

- *Derive* measure of `(max 0.5 (random))` as

$$P_m\ B = P\ (f^{-1}\ B)$$

where $f^{-1}\ B\ =\ \{r \in [0, 1] \mid f\ r \in B\}$

- Factored into random and deterministic parts:

$$P_m = P \circ f^{-1}$$

Probability Measures Via Preimages

- Interpret `(max 0.5 (random))` as $f : [0, 1] \to \mathbb{R}$, defined

$$f\ r\ =\ \max(0.5, r)$$

- *Derive* measure of `(max 0.5 (random))` as

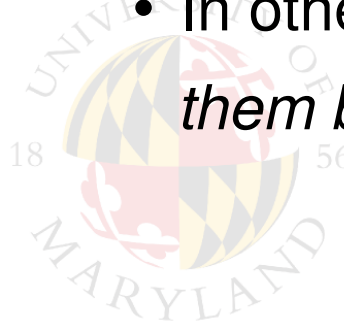
$$P_m\ B = P\ (f^{-1}\ B)$$

where $f^{-1}\ B\ =\ \{r \in [0, 1] \mid f\ r \in B\}$

- Factored into random and deterministic parts:

$$P_m = P \circ f^{-1}$$

- In other words, compute measures of expressions by *running them backwards*



Crazy Idea is Feasible If...

- Seems like we need:

Crazy Idea is Feasible If...

- Seems like we need:
  - Standard interpretation of programs as pure functions from a random source

Crazy Idea is Feasible If...

- Seems like we need:
  - Standard interpretation of programs as pure functions from a random source
  - Efficient way to compute preimage sets

Crazy Idea is Feasible If...

- Seems like we need:
  - Standard interpretation of programs as pure functions from a random source
  - Efficient way to compute preimage sets
  - Efficient representation of arbitrary sets

Crazy Idea is Feasible If...

- Seems like we need:
  - Standard interpretation of programs as pure functions from a random source
  - Efficient way to compute preimage sets
  - Efficient representation of arbitrary sets
  - Efficient way to compute areas of preimage sets

Crazy Idea is Feasible If...

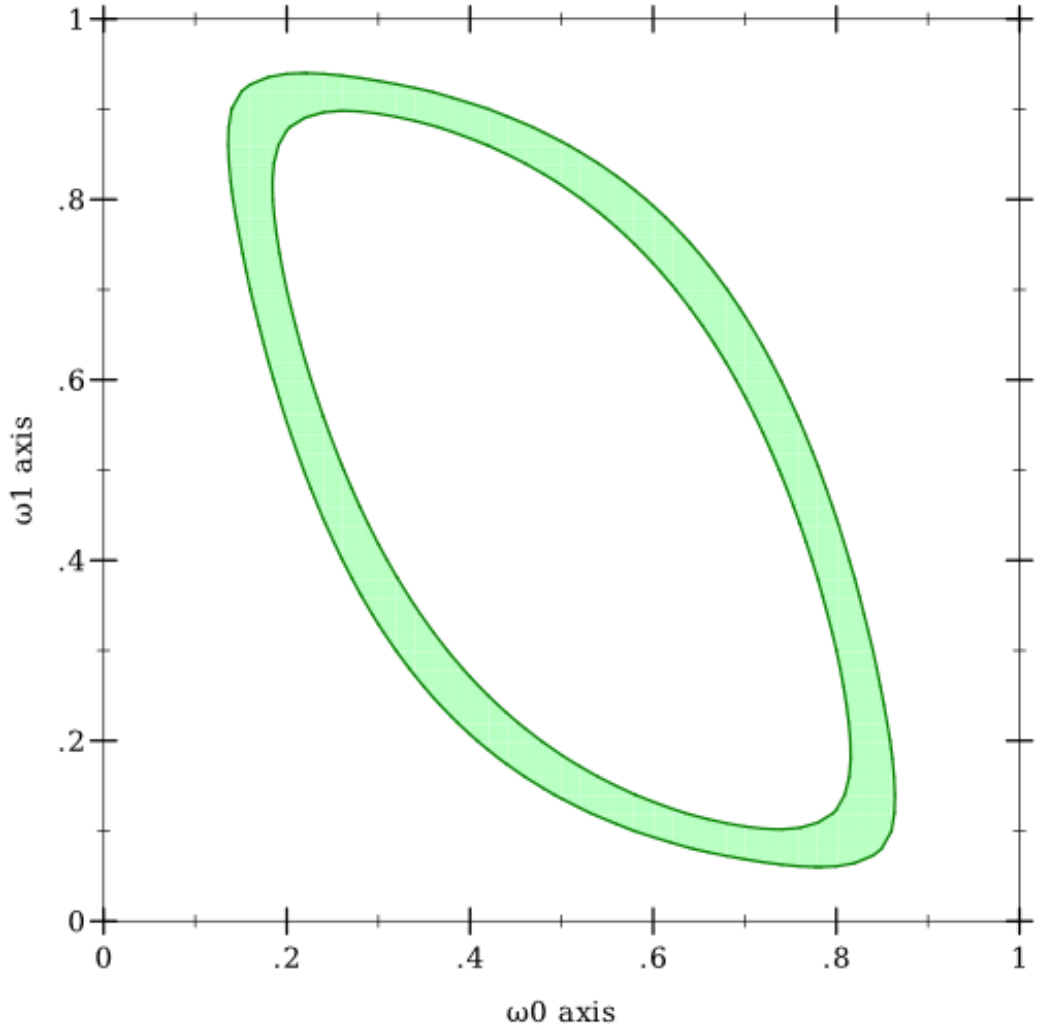
- Seems like we need:
  - Standard interpretation of programs as pure functions from a random source
  - Efficient way to compute preimage sets
  - Efficient representation of arbitrary sets
  - Efficient way to compute areas of preimage sets
  - Proof of correctness w.r.t. standard interpretation

Crazy Idea is Feasible If...

- Seems like we need:
  - Standard interpretation of programs as pure functions from a random source
  - Efficient way to compute preimage sets
  - Efficient representation of arbitrary sets
  - Efficient way to compute areas of preimage sets
  - Proof of correctness w.r.t. standard interpretation
- WAT

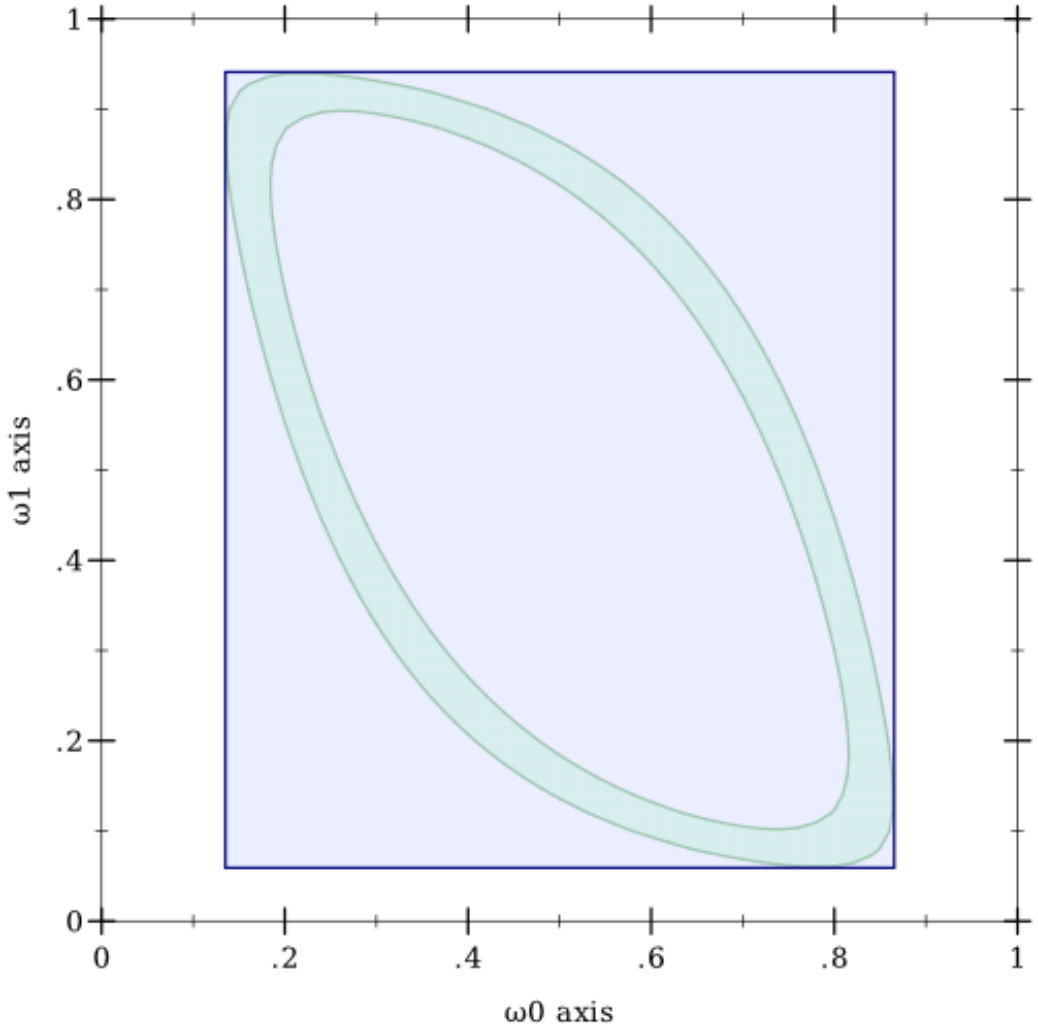
What About Approximating?

Conservative approximation with rectangles:



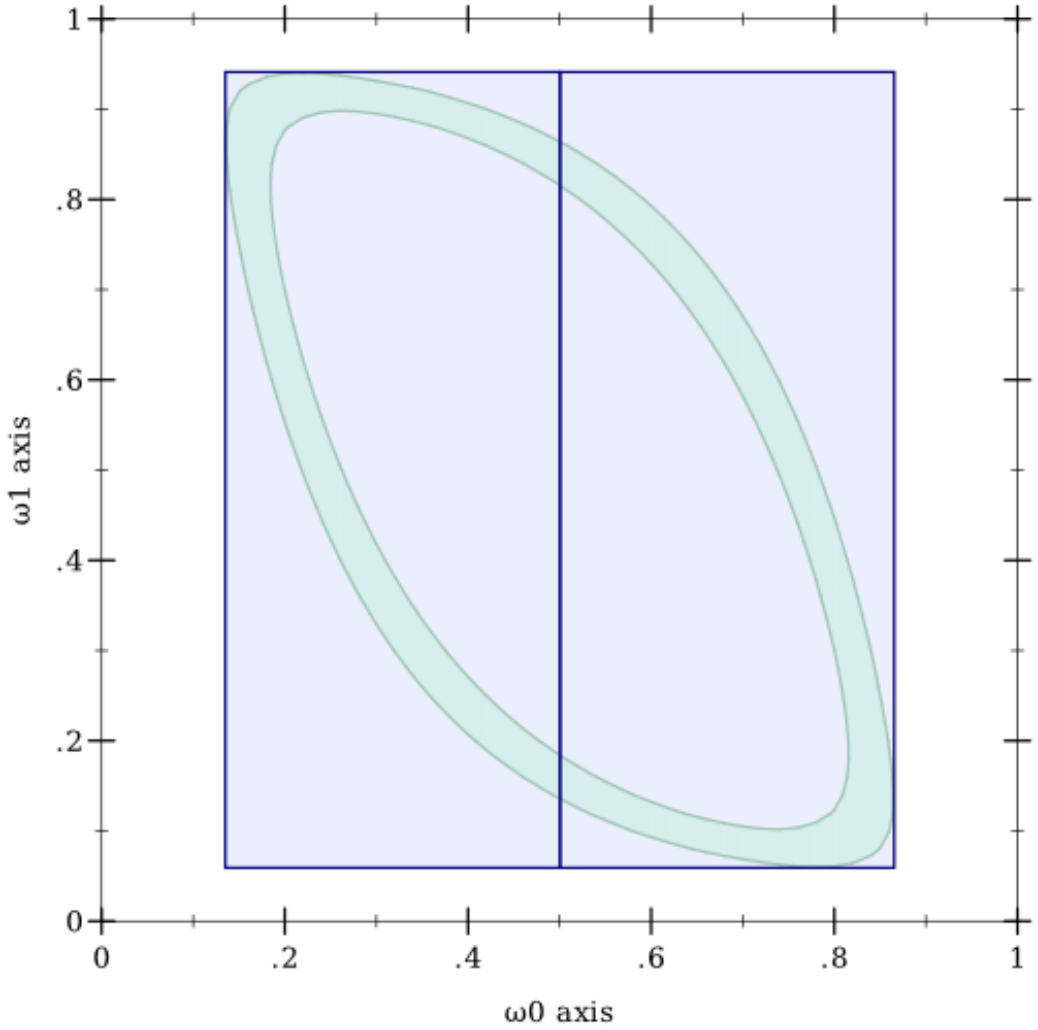
What About Approximating?

Conservative approximation with rectangles:



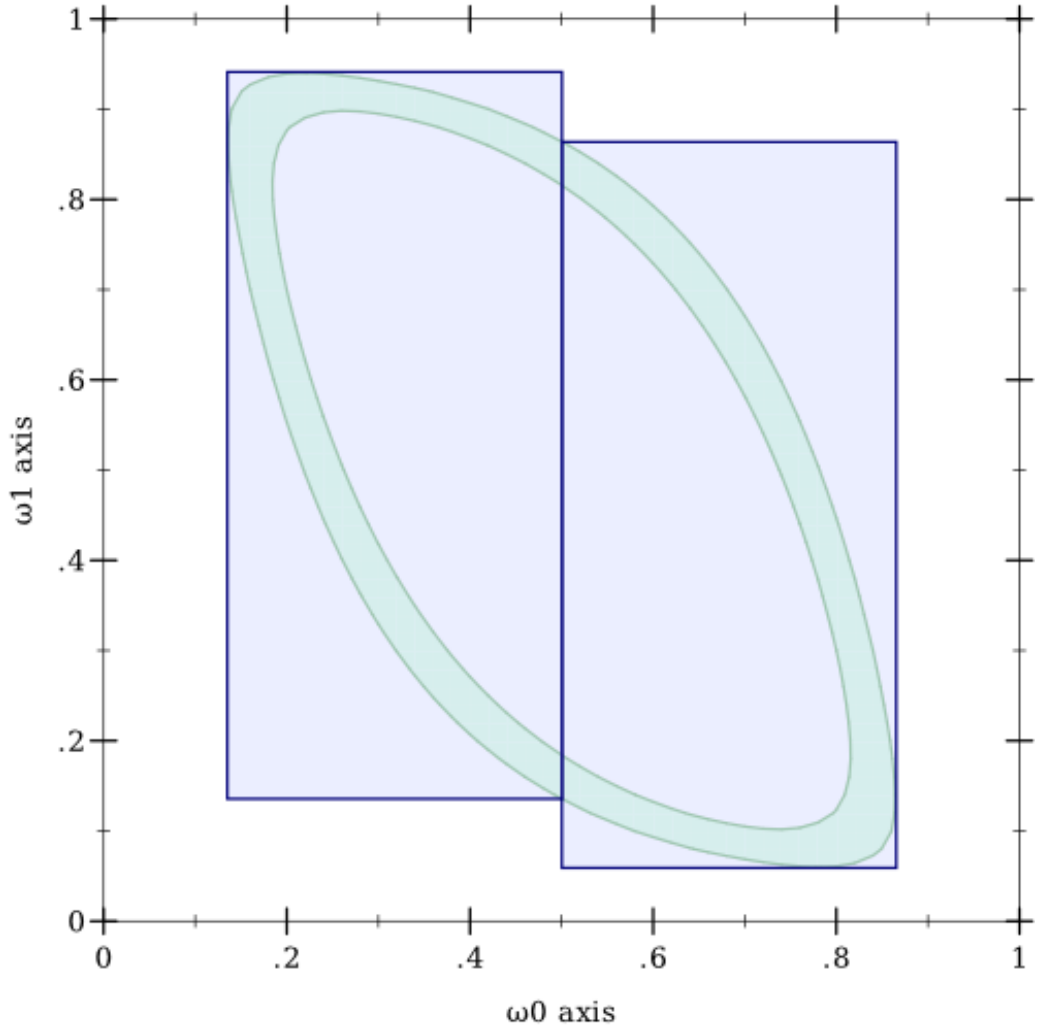
What About Approximating?

Restricting preimages to rectangular subdomains:



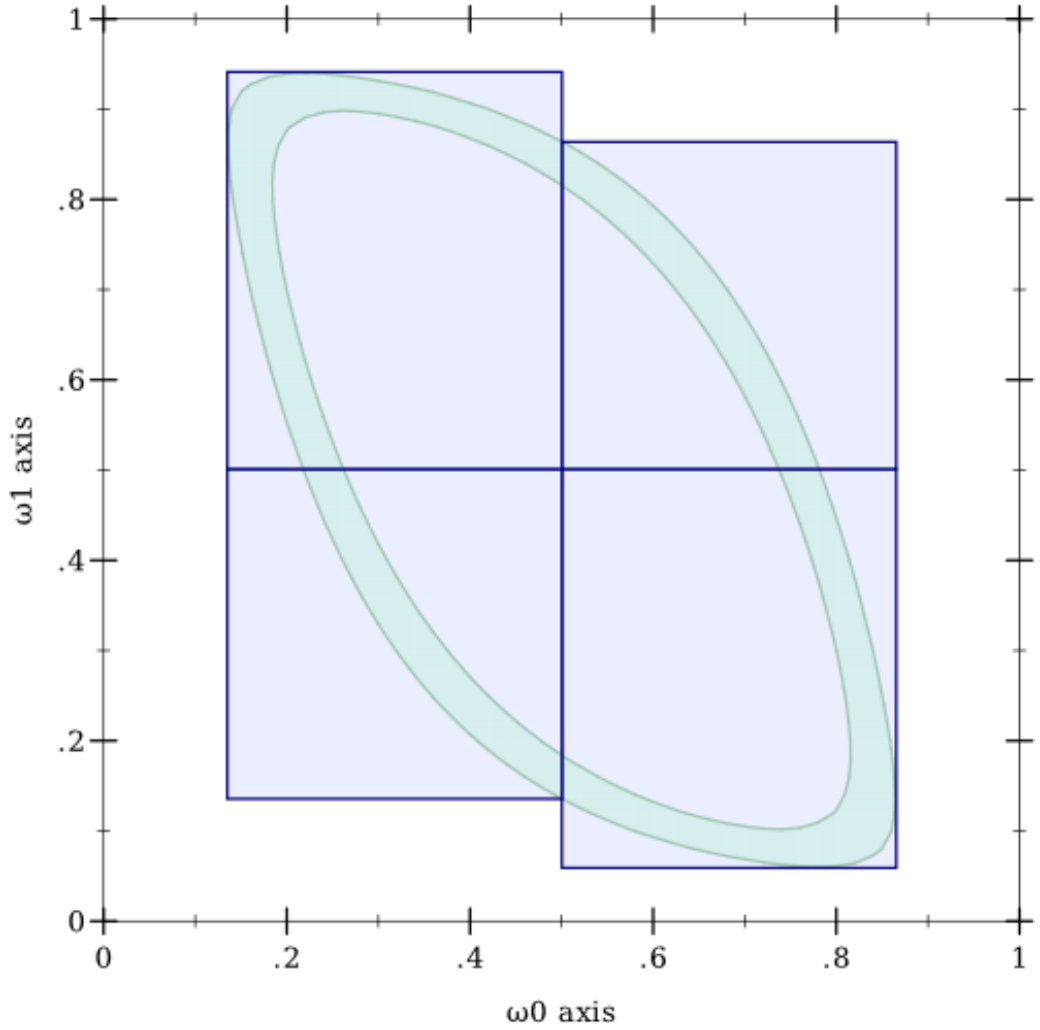
What About Approximating?

Restricting preimages to rectangular subdomains:



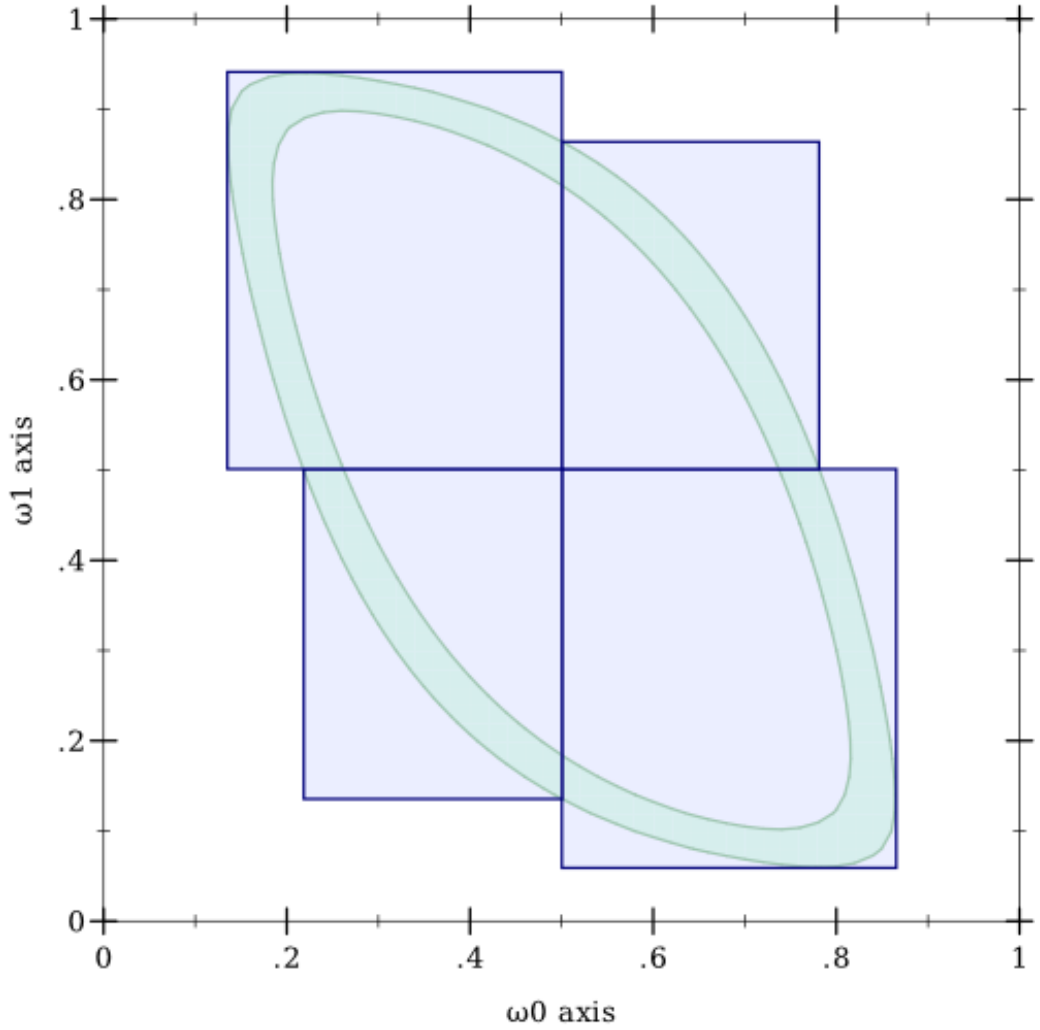
What About Approximating?

Restricting preimages to rectangular subdomains:



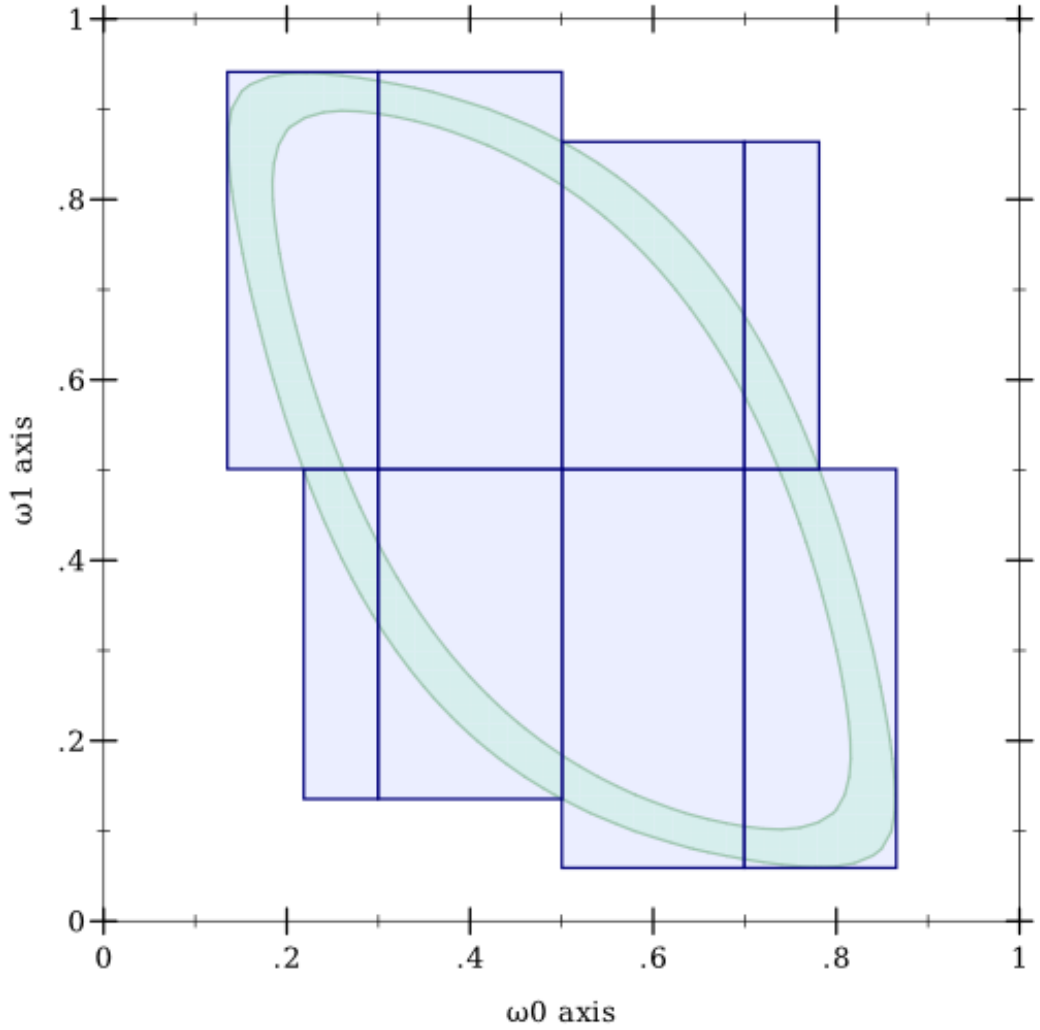
What About Approximating?

Restricting preimages to rectangular subdomains:



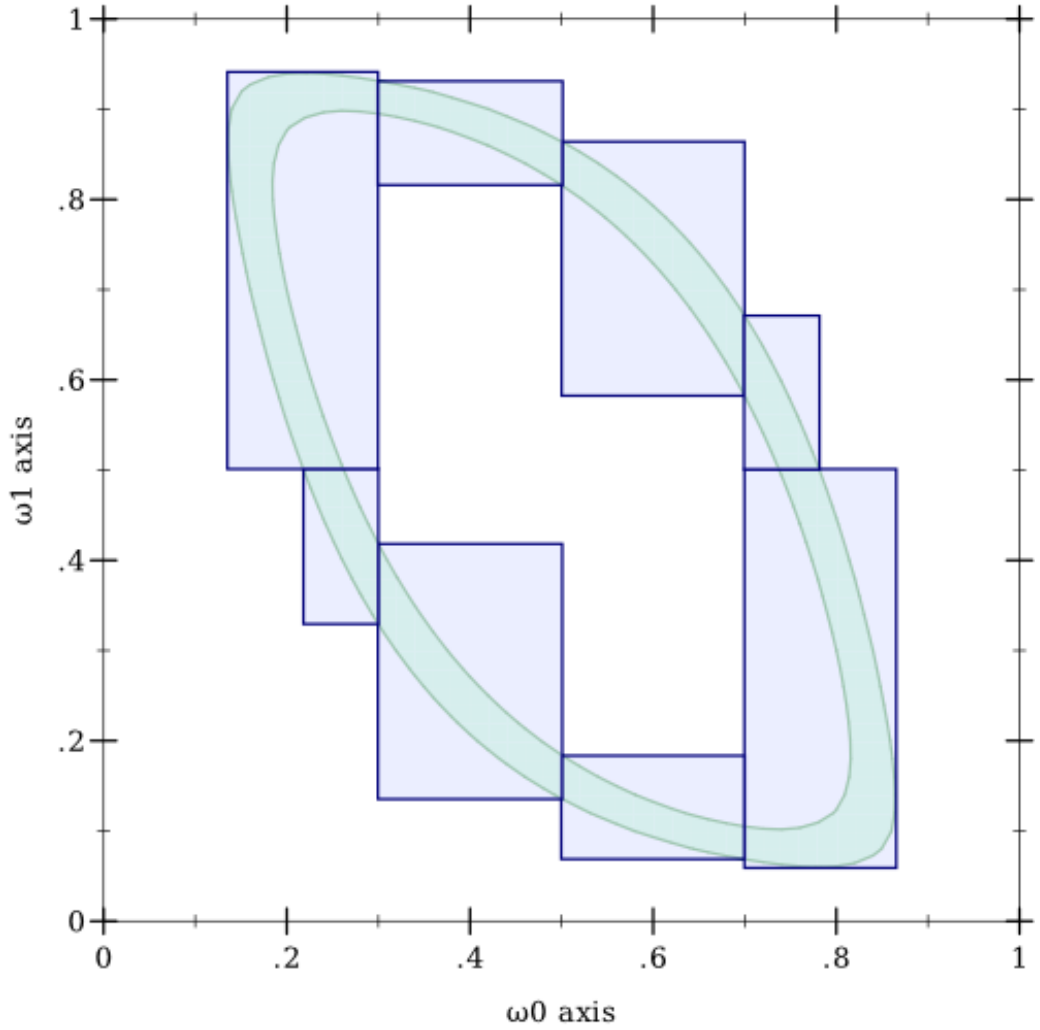
What About Approximating?

Restricting preimages to rectangular subdomains:



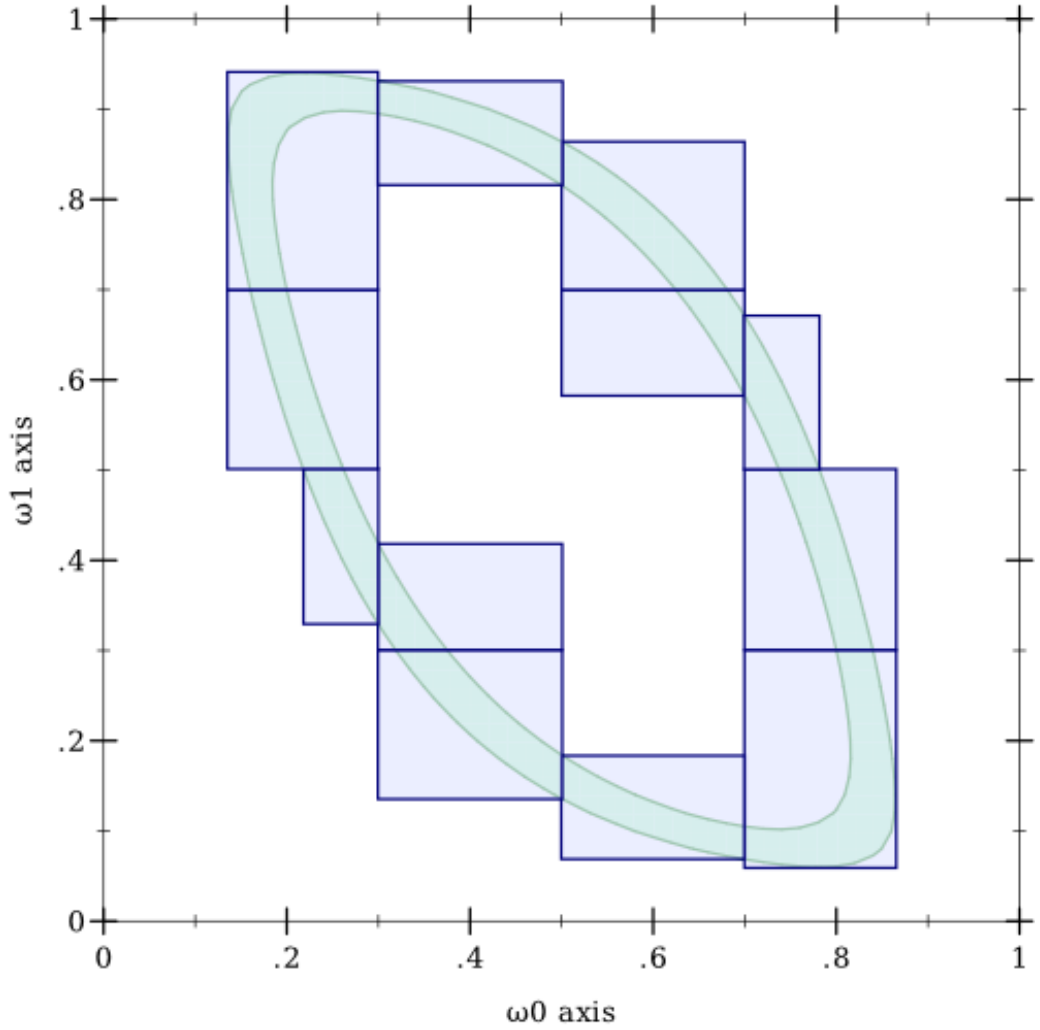
What About Approximating?

Restricting preimages to rectangular subdomains:



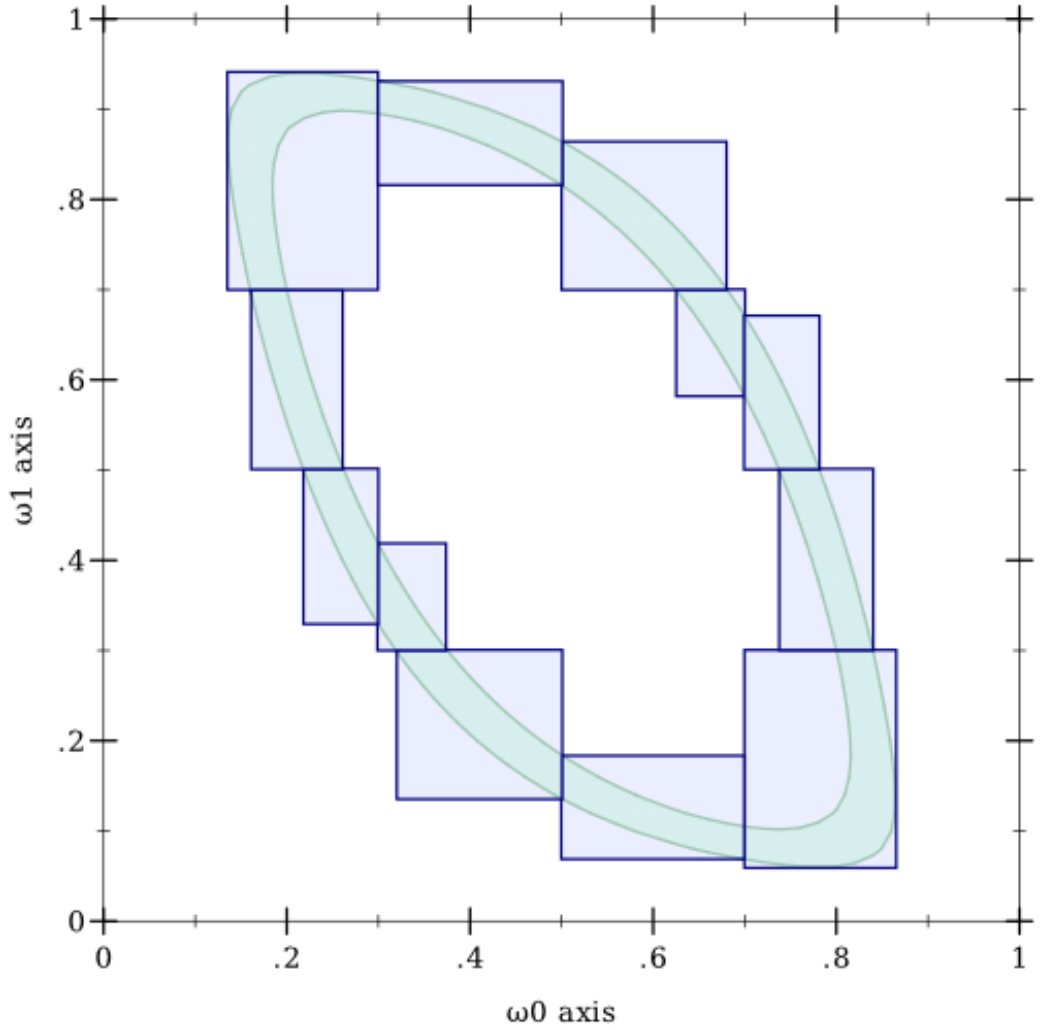
What About Approximating?

Restricting preimages to rectangular subdomains:



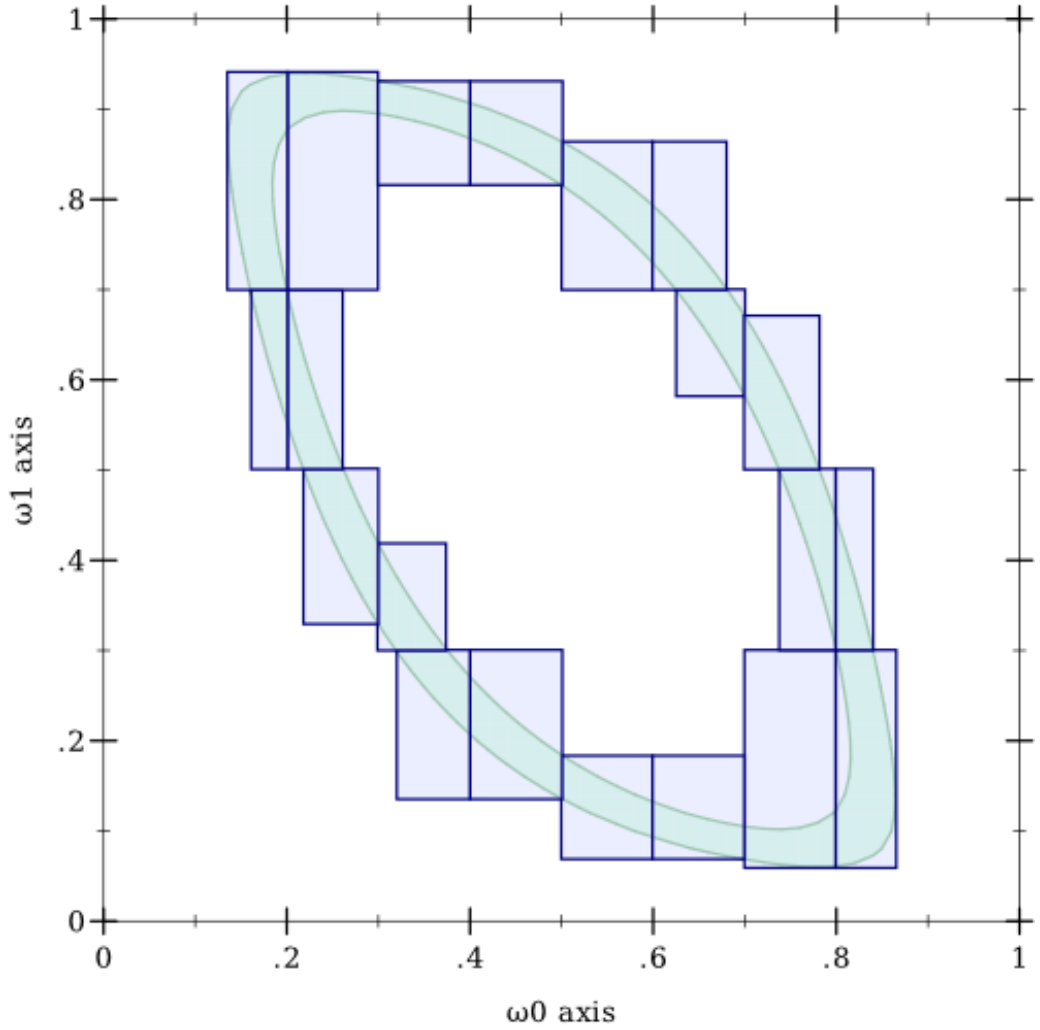
What About Approximating?

Restricting preimages to rectangular subdomains:



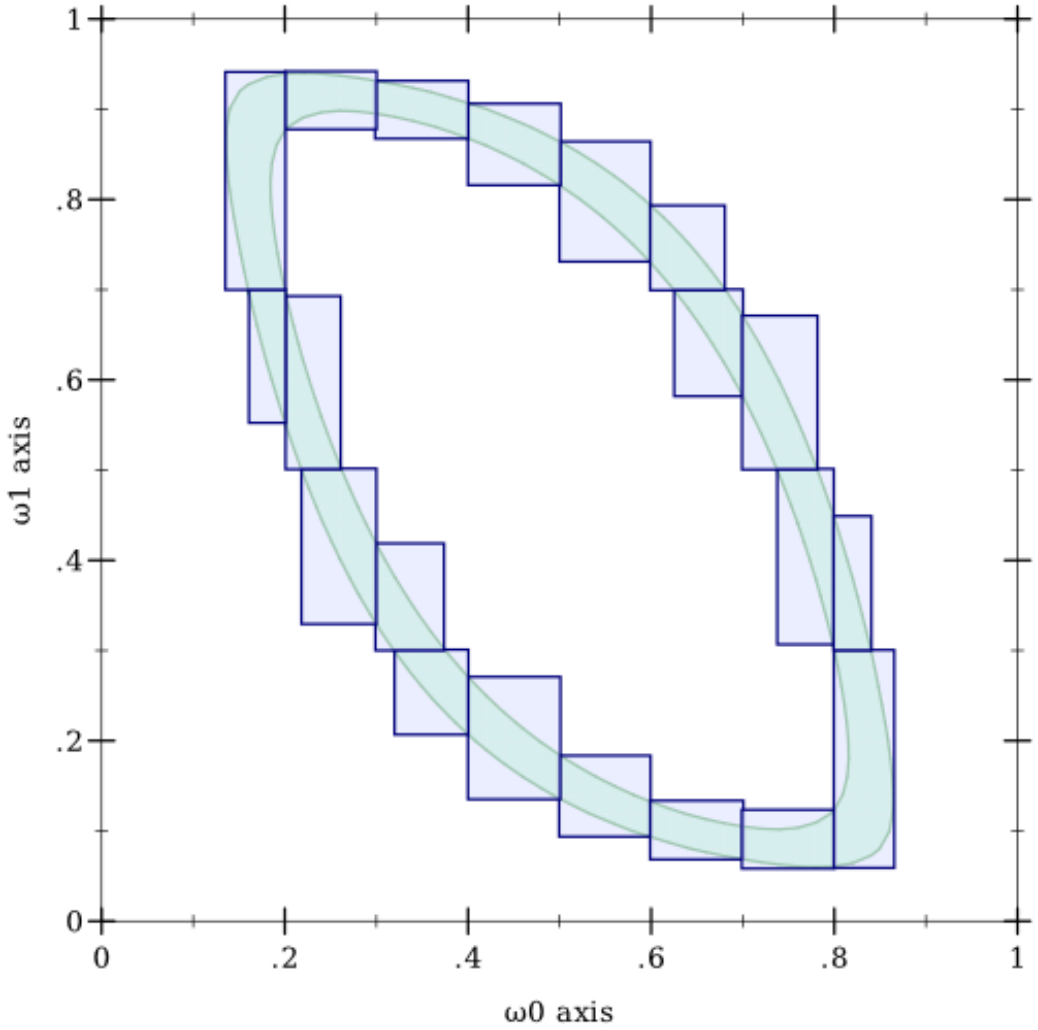
What About Approximating?

Restricting preimages to rectangular subdomains:



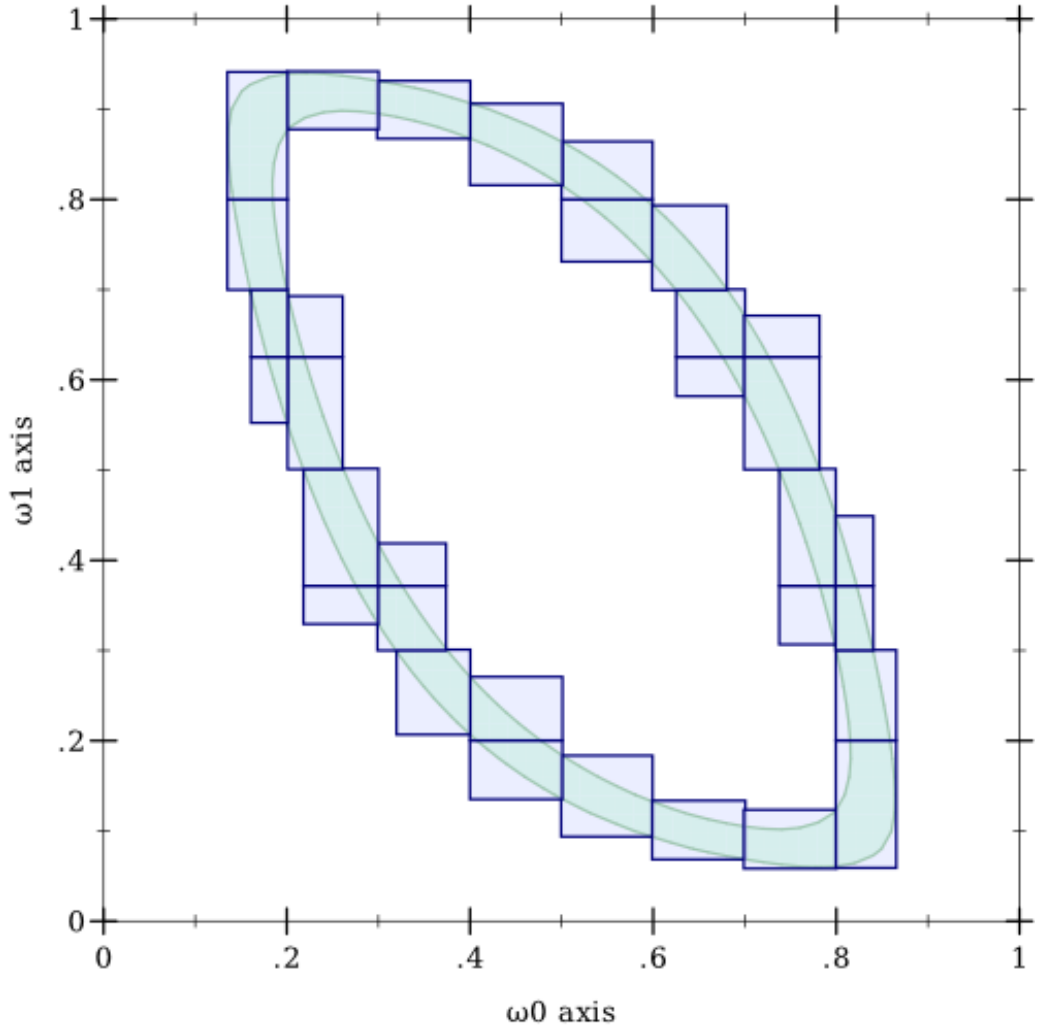
What About Approximating?

Restricting preimages to rectangular subdomains:



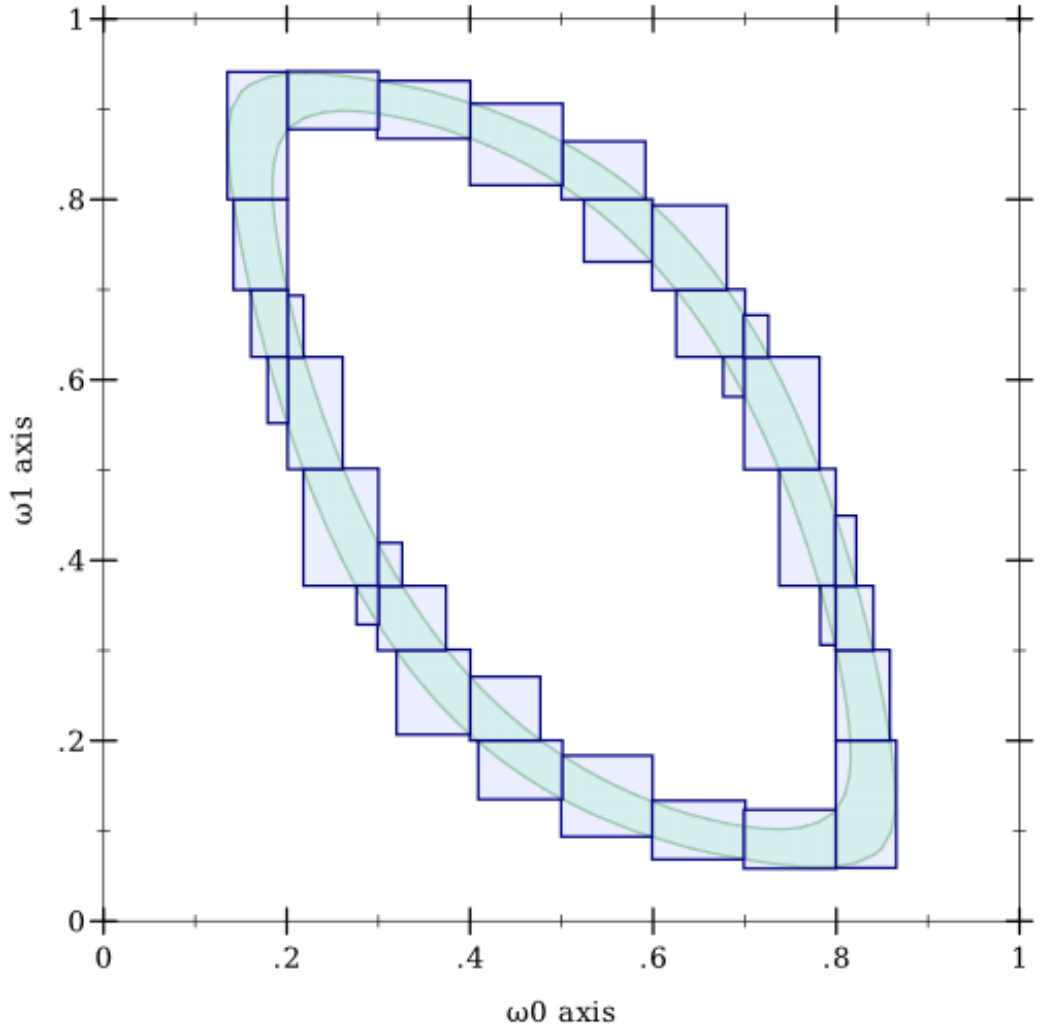
What About Approximating?

Restricting preimages to rectangular subdomains:



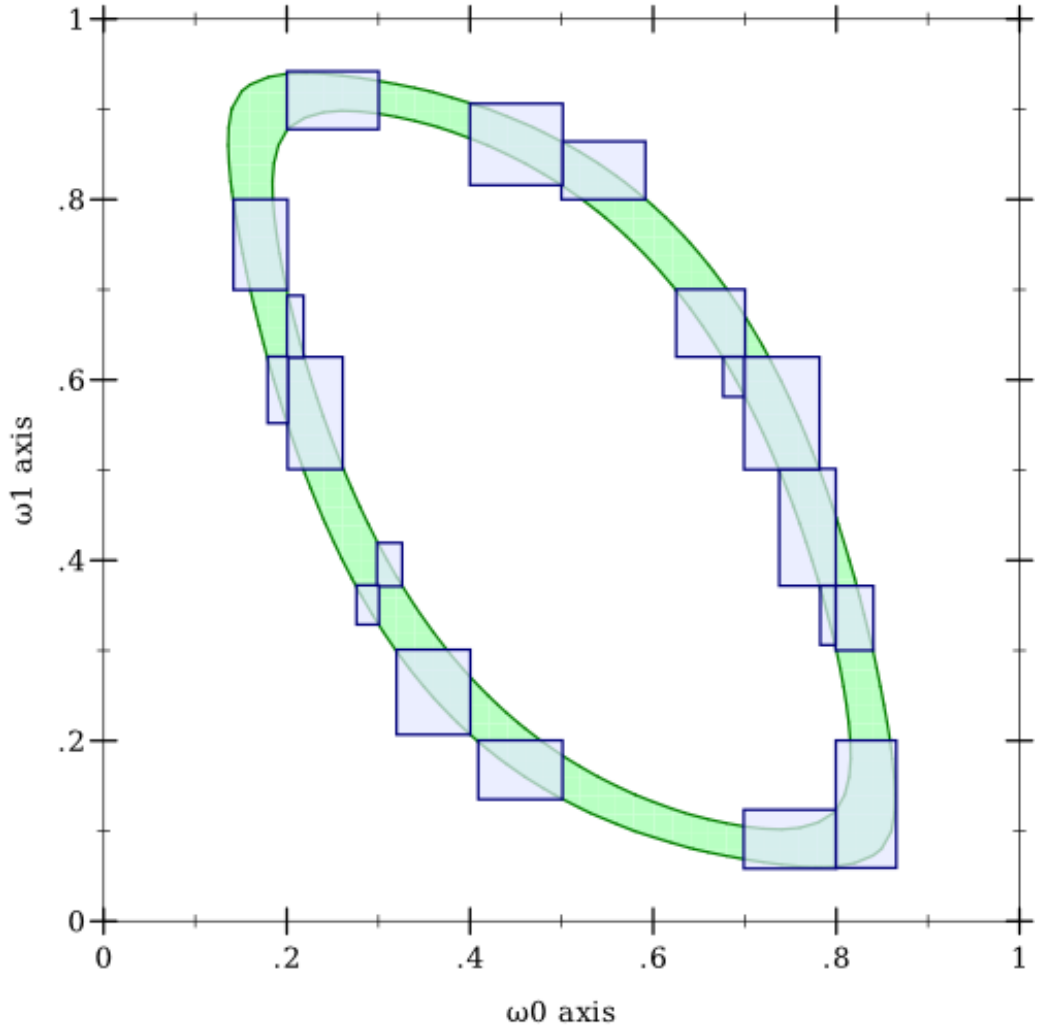
What About Approximating?

Restricting preimages to rectangular subdomains:



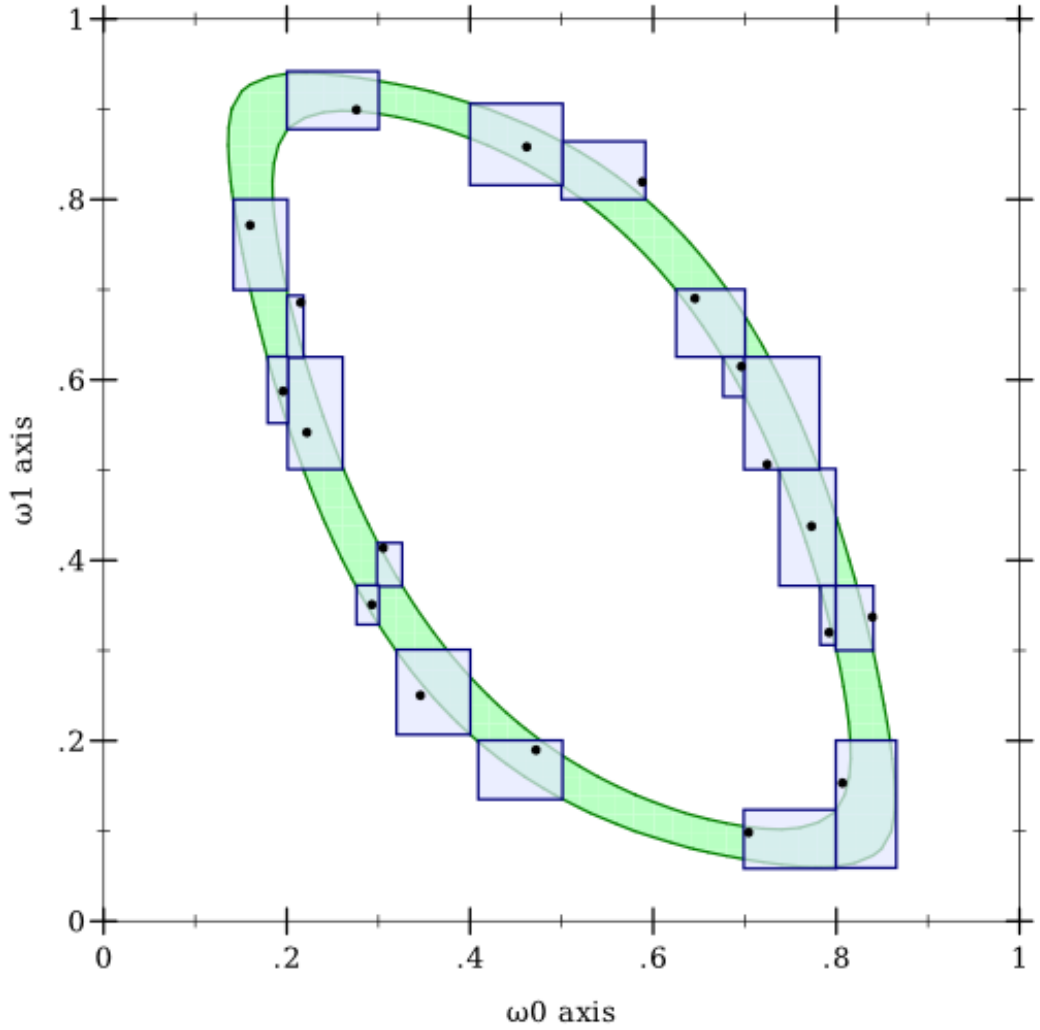
What About Approximating?

Sampling: exponential to quadratic (e.g. days to minutes)



What About Approximating?

Sampling: exponential to quadratic (e.g. days to minutes)



Contribution: Making This Crazy Idea Feasible

- Standard interpretation of programs as pure functions from a random source
- Efficient way to compute preimage sets
- Efficient representation of arbitrary sets
- Efficient way to compute volumes of preimage sets

- Proof of correctness w.r.t. standard interpretation

Contribution: Making This Crazy Idea Feasible

- Standard interpretation of programs as pure functions from a random source
- Efficient way to compute abstract preimage subsets
- Efficient representation of arbitrary sets
- Efficient way to compute volumes of preimage sets

- Proof of correctness w.r.t. standard interpretation

Contribution: Making This Crazy Idea Feasible

- Standard interpretation of programs as pure functions from a random source
- Efficient way to compute abstract preimage subsets
- Efficient representation of abstract sets
- Efficient way to compute volumes of preimage sets

- Proof of correctness w.r.t. standard interpretation

Contribution: Making This Crazy Idea Feasible

- Standard interpretation of programs as pure functions from a random source
- Efficient way to compute abstract preimage subsets
- Efficient representation of abstract sets
- Efficient way to sample uniformly in preimage sets

- Proof of correctness w.r.t. standard interpretation

Contribution: Making This Crazy Idea Feasible

- Standard interpretation of programs as pure functions from a random source
- Efficient way to compute abstract preimage subsets
- Efficient representation of abstract sets
- Efficient way to sample uniformly in preimage sets
  - Efficient domain partition sampling
- Proof of correctness w.r.t. standard interpretation

Contribution: Making This Crazy Idea Feasible

- Standard interpretation of programs as pure functions from a random source
- Efficient way to compute abstract preimage subsets
- Efficient representation of abstract sets
- Efficient way to sample uniformly in preimage sets
  - Efficient domain partition sampling
  - Efficient way to determine whether a domain sample is actually in the preimage (just use standard interpretation)
- Proof of correctness w.r.t. standard interpretation

How Many Meanings?

- Start with pure programs, then lift by threading a random store

How Many Meanings?

- Start with pure programs, then lift by threading a random store
- Nonrecursive, nonprobabilistic programs: $[\![\cdot]\!]$, $[\![\cdot]\!]_{\mathrm{pre}}$, $[\![\cdot]\!]_{\widehat{\mathrm{pre}}}$

How Many Meanings?

- Start with pure programs, then lift by threading a random store
- Nonrecursive, nonprobabilistic programs: $[\![\cdot]\!]$, $[\![\cdot]\!]_{\mathrm{pre}}$, $[\![\cdot]\!]_{\widehat{\mathrm{pre}}}$
- Add 3 semantic functions for recursion and probabilistic choice

How Many Meanings?

- Start with pure programs, then lift by threading a random store
- Nonrecursive, nonprobabilistic programs: $[\![\cdot]\!]$, $[\![\cdot]\!]_{\mathrm{pre}}$, $[\![\cdot]\!]_{\widehat{\mathrm{pre}}}$
- Add 3 semantic functions for recursion and probabilistic choice
- Full development needs 2 more to transfer theorems from measure theory...

How Many Meanings?

- Start with pure programs, then lift by threading a random store
- Nonrecursive, nonprobabilistic programs: $[\![\cdot]\!], [\![\cdot]\!]_{\mathrm{pre}}, [\![\cdot]\!]_{\widehat{\mathrm{pre}}}$
- Add 3 semantic functions for recursion and probabilistic choice
- Full development needs 2 more to transfer theorems from measure theory...
- ... oh, and 1 more to collect information for Monte Carlo integration

How Many Meanings?

- Start with pure programs, then lift by threading a random store
- Nonrecursive, nonprobabilistic programs: $[\![\cdot]\!]$, $[\![\cdot]\!]_{\mathrm{pre}}$, $[\![\cdot]\!]_{\widehat{\mathrm{pre}}}$
- Add 3 semantic functions for recursion and probabilistic choice
- Full development needs 2 more to transfer theorems from measure theory...
- ... oh, and 1 more to collect information for Monte Carlo integration

Tally: 3+3+2+1 = 9 semantic functions, 11 or 12 rules each

Enter Category Theory

- Moggi (1989): Introduces monads for interpreting effects

Enter Category Theory

- Moggi (1989): Introduces monads for interpreting effects
- Other kinds of categories: idioms, arrows

Enter Category Theory

- Moggi (1989): Introduces monads for interpreting effects
- Other kinds of categories: idioms, arrows
- Arrow defined by type constructor $x \rightsquigarrow_a y$ and these combinators:

$$\begin{aligned}
\mathtt{arr}_a &: (x \to y) \to (x \rightsquigarrow_a y) \\
(\ggg_a) &: (x \rightsquigarrow_a y) \to (y \rightsquigarrow_a z) \to (x \rightsquigarrow_a z) \\
(\&\&\&_a) &: (x \rightsquigarrow_a y) \to (x \rightsquigarrow_a z) \to (x \rightsquigarrow_a \langle y, z \rangle) \\
\mathtt{ifte}_a &: (x \rightsquigarrow_a \mathtt{Bool}) \to (x \rightsquigarrow_a y) \to (x \rightsquigarrow_a y) \to (x \rightsquigarrow_a y) \\
\mathtt{lazy}_a &: (1 \to (x \rightsquigarrow_a y)) \to (x \rightsquigarrow_a y)
\end{aligned}$$

Enter Category Theory

- Moggi (1989): Introduces monads for interpreting effects
- Other kinds of categories: idioms, arrows
- Arrow defined by type constructor $x \rightsquigarrow_a y$ and these combinators:

$$\begin{aligned}
\mathtt{arr}_a &: (x \to y) \to (x \rightsquigarrow_a y) \\
(\ggg_a) &: (x \rightsquigarrow_a y) \to (y \rightsquigarrow_a z) \to (x \rightsquigarrow_a z) \\
(\&\&\&_a) &: (x \rightsquigarrow_a y) \to (x \rightsquigarrow_a z) \to (x \rightsquigarrow_a \langle y, z \rangle) \\
\mathtt{ifte}_a &: (x \rightsquigarrow_a \mathtt{Bool}) \to (x \rightsquigarrow_a y) \to (x \rightsquigarrow_a y) \to (x \rightsquigarrow_a y) \\
\mathtt{lazy}_a &: (1 \to (x \rightsquigarrow_a y)) \to (x \rightsquigarrow_a y)
\end{aligned}$$

- Arrows are always *function-like*

Reducing Complexity

Function arrow: $x \rightsquigarrow y$ is just $x \to y$

Reducing Complexity

Function arrow: $x \rightsquigarrow y$ is just $x \to y$

$$\texttt{arr}\ f = f$$

Reducing Complexity

Function arrow: $x \rightsquigarrow y$ is just $x \to y$

$$\texttt{arr}\ f \ = \ f$$

$$f_1 \ggg f_2 \ = \ \lambda r.\, f_2\ (f_1\ r)$$

Reducing Complexity

Function arrow: $x \rightsquigarrow y$ is just $x \to y$

$$\texttt{arr}\ f \ = \ f$$

$$f_1 \ggg f_2 \ = \ \lambda r.\, f_2\ (f_1\ r)$$

$$[\![\texttt{fst}\ e]\!] \ = \ [\![e]\!] \ggg \texttt{arr fst}$$

Reducing Complexity

Function arrow: $x \rightsquigarrow y$ is just $x \to y$

$$\texttt{arr}\ f = f$$

$$f_1 \ggg f_2 = \lambda r.\, f_2\ (f_1\ r)$$

$$[\![\texttt{fst}\ e]\!] = [\![e]\!] \ggg \texttt{arr fst}$$

$$f_1 \mathbin{\&\&\&} f_2 = \lambda r.\, \langle f_1\ r, f_2\ r \rangle$$

Reducing Complexity

Function arrow: $x \rightsquigarrow y$ is just $x \to y$

$$\texttt{arr}\ f \ = \ f$$

$$f_1 \ggg f_2 \ = \ \lambda r.\, f_2\ (f_1\ r)$$

$$[\![\texttt{fst}\ e]\!] \ = \ [\![e]\!] \ggg \texttt{arr fst}$$

$$f_1 \mathbin{\&\&\&} f_2 \ = \ \lambda r.\, \langle f_1\ r, f_2\ r \rangle$$

$$[\![\langle e_1, e_2 \rangle]\!] \ = \ [\![e_1]\!] \mathbin{\&\&\&} [\![e_2]\!]$$

Reducing Complexity

Function arrow: $x \rightsquigarrow y$ is just $x \to y$

$$\texttt{arr}\ f \ = \ f$$

$$f_1 \ggg f_2 \ = \ \lambda r.\, f_2\ (f_1\ r)$$

$$[\![\texttt{fst}\ e]\!] \ = \ [\![e]\!] \ggg \texttt{arr fst}$$

$$f_1 \mathbin{\&\&\&} f_2 \ = \ \lambda r.\, \langle f_1\ r, f_2\ r \rangle$$

$$[\![\langle e_1, e_2 \rangle]\!] \ = \ [\![e_1]\!] \mathbin{\&\&\&} [\![e_2]\!]$$

$$[\![\texttt{let}\ e\ e_b]\!] \ = \ ([\![e]\!] \mathbin{\&\&\&} \texttt{arr id}) \ggg [\![e_b]\!]$$

Reducing Complexity

Function arrow: $x \rightsquigarrow y$ is just $x \to y$

$$\texttt{arr}\ f \ = \ f$$

$$f_1 \ggg f_2 \ = \ \lambda r.\, f_2\ (f_1\ r)$$

$$[\![\texttt{fst}\ e]\!] \ = \ [\![e]\!] \ggg \texttt{arr fst}$$

$$f_1 \mathbin{\&\&\&} f_2 \ = \ \lambda r.\, \langle f_1\ r, f_2\ r \rangle$$

$$[\![\langle e_1, e_2 \rangle]\!] \ = \ [\![e_1]\!] \mathbin{\&\&\&} [\![e_2]\!]$$

$$[\![\texttt{let}\ e\ e_b]\!] \ = \ ([\![e]\!] \mathbin{\&\&\&} \texttt{arr id}) \ggg [\![e_b]\!]$$

$$[\![\texttt{env}\ 0]\!] \ = \ \texttt{arr fst}$$

Reducing Complexity

Function arrow: $x \rightsquigarrow y$ is just $x \to y$

$$\texttt{arr}\ f \ = \ f$$

$$f_1 \ggg f_2 \ = \ \lambda r.\, f_2\ (f_1\ r)$$

$$[\![\texttt{fst}\ e]\!] \ = \ [\![e]\!] \ggg \texttt{arr fst}$$

$$f_1 \,\&\&\&\, f_2 \ = \ \lambda r.\, \langle f_1\ r, f_2\ r \rangle$$

$$[\![\langle e_1, e_2 \rangle]\!] \ = \ [\![e_1]\!] \,\&\&\&\, [\![e_2]\!]$$

$$[\![\texttt{let}\ e\ e_b]\!] \ = \ ([\![e]\!] \,\&\&\&\, \texttt{arr id}) \ggg [\![e_b]\!]$$

$$[\![\texttt{env}\ 0]\!] \ = \ \texttt{arr fst}$$

$$[\![\texttt{env}\ (n+1)]\!] \ = \ \texttt{arr snd} \ggg [\![\texttt{env}\ n]\!]$$

Pair Preimages

$$f_1 = \lambda r.\, \texttt{fst}\ r + \texttt{snd}\ r \qquad f_2 = \lambda r.\, \texttt{fst}\ r \cdot \texttt{snd}\ r$$

Pair Preimages

$$f_1 = \lambda r.\, \texttt{fst}\ r + \texttt{snd}\ r \qquad f_2 = \lambda r.\, \texttt{fst}\ r \cdot \texttt{snd}\ r$$

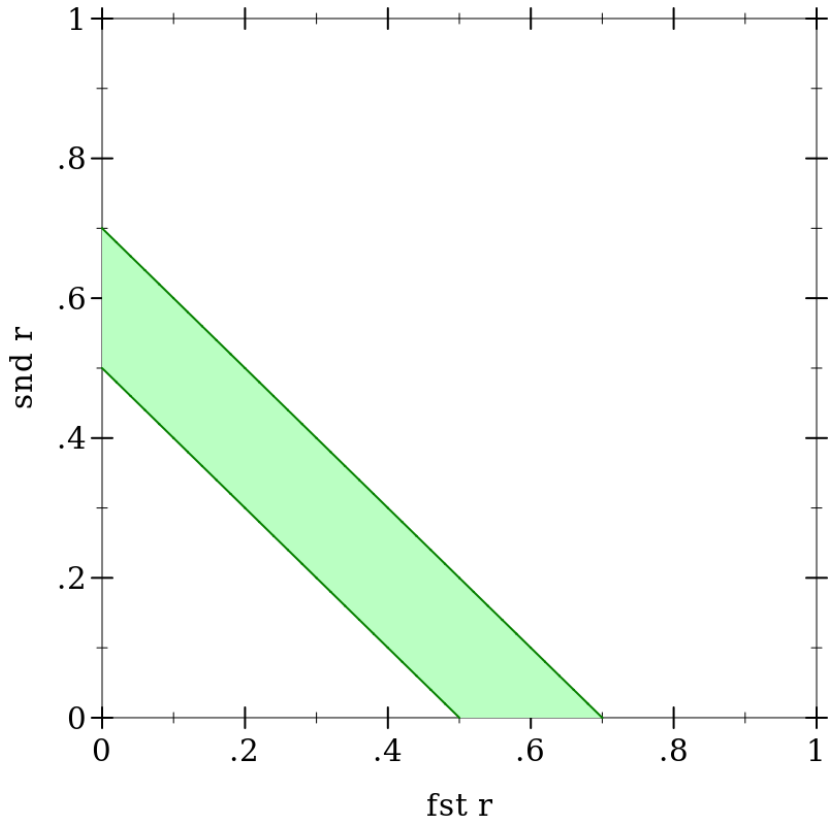
$$f = f_1 \,\&\&\&\, f_2 = \lambda r.\, \langle \texttt{fst}\ r + \texttt{snd}\ r, \texttt{fst}\ r \cdot \texttt{snd}\ r \rangle$$

Pair Preimages

$$f_1 = \lambda r.\, \texttt{fst}\ r + \texttt{snd}\ r \qquad f_2 = \lambda r.\, \texttt{fst}\ r \cdot \texttt{snd}\ r$$

$$f = f_1 \,\&\&\&\, f_2 = \lambda r.\, \langle \texttt{fst}\ r + \texttt{snd}\ r, \texttt{fst}\ r \cdot \texttt{snd}\ r \rangle$$

$f_1^{-1}([0.5, 0.7])$:

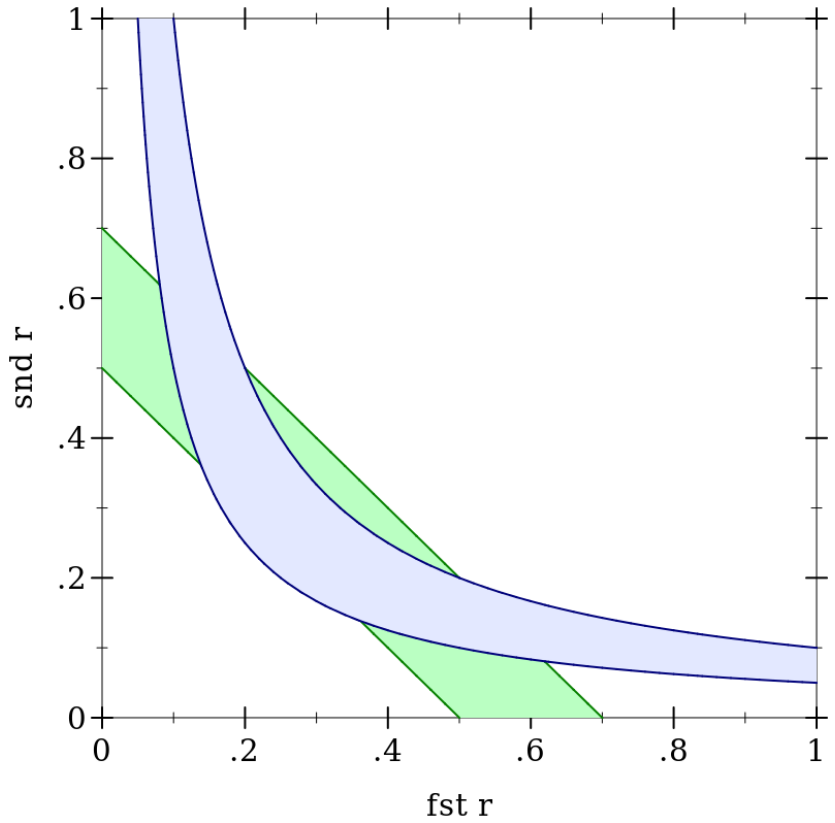


Pair Preimages

$$f_1 = \lambda r.\, \texttt{fst}\ r + \texttt{snd}\ r \qquad f_2 = \lambda r.\, \texttt{fst}\ r \cdot \texttt{snd}\ r$$

$$f = f_1 \,\&\&\&\, f_2 = \lambda r.\, \langle \texttt{fst}\ r + \texttt{snd}\ r, \texttt{fst}\ r \cdot \texttt{snd}\ r \rangle$$

$f_1^{-1}([0.5, 0.7])$ and $f_2^{-1}([0.05, 0.1])$:

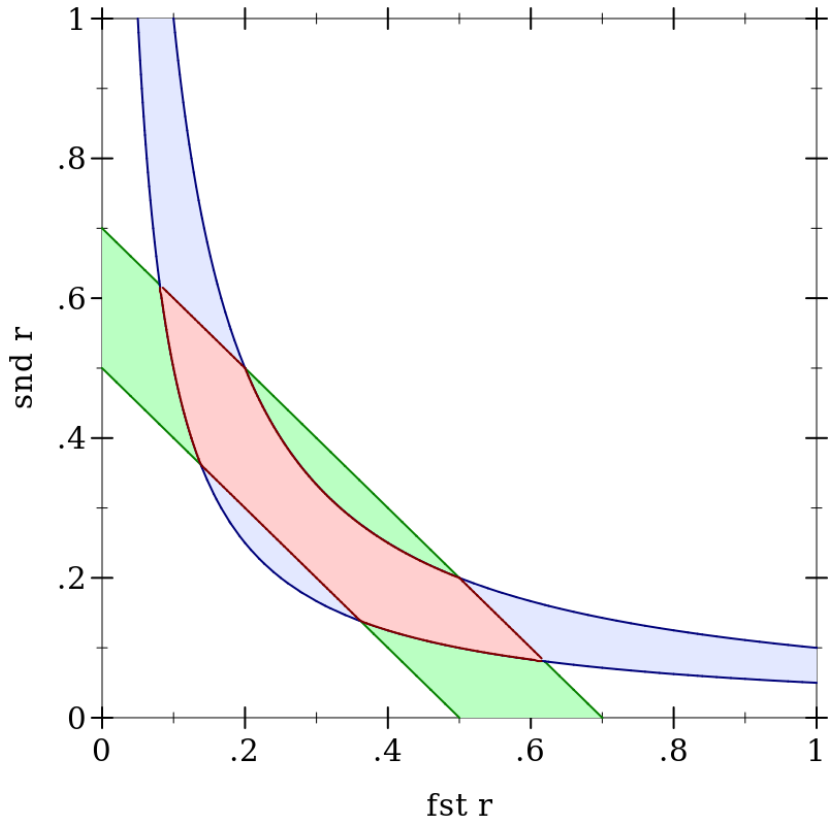


Pair Preimages

$$f_1 = \lambda r.\, \texttt{fst}\ r + \texttt{snd}\ r \qquad f_2 = \lambda r.\, \texttt{fst}\ r \cdot \texttt{snd}\ r$$

$$f = f_1 \,\&\&\&\, f_2 = \lambda r.\, \langle \texttt{fst}\ r + \texttt{snd}\ r, \texttt{fst}\ r \cdot \texttt{snd}\ r \rangle$$

$f^{-1}([0.5, 0.7] \times [0.05, 0.1])$:



Correctness Theorems For Low, Low Prices

- Define $\texttt{lift}_{\text{pre}}\ f = f^{-1}$

Correctness Theorems For Low, Low Prices

- Define $\texttt{lift}_{\text{pre}}\ f = f^{-1}$
- Derive $(\&\&\&_{\text{pre}})$ and others so that $\texttt{lift}_{\text{pre}}$ distributes; e.g.

$$\texttt{lift}_{\text{pre}}\ (f_1\ \&\&\&\ f_2) = (\texttt{lift}_{\text{pre}}\ f_1)\ \&\&\&_{\text{pre}}\ (\texttt{lift}_{\text{pre}}\ f_2)$$

Correctness Theorems For Low, Low Prices

- Define $\texttt{lift}_{\text{pre}}\ f = f^{-1}$
- Derive $(\&\&\&_{\text{pre}})$ and others so that $\texttt{lift}_{\text{pre}}$ distributes; e.g.

$$\texttt{lift}_{\text{pre}}\ (f_1\ \&\&\&\ f_2) = (\texttt{lift}_{\text{pre}}\ f_1)\ \&\&\&_{\text{pre}}\ (\texttt{lift}_{\text{pre}}\ f_2)$$

- Distributive properties makes proving this very easy:

Theorem (correctness). For all e, $[\![e]\!]_{\text{pre}} = \texttt{lift}_{\text{pre}}\ [\![e]\!]$.

Correctness Theorems For Low, Low Prices

- Define $\texttt{lift}_{\text{pre}}\ f = f^{-1}$
- Derive $(\&\&\&_{\text{pre}})$ and others so that $\texttt{lift}_{\text{pre}}$ distributes; e.g.

$$\texttt{lift}_{\text{pre}}\ (f_1\ \&\&\&\ f_2)\ =\ (\texttt{lift}_{\text{pre}}\ f_1)\ \&\&\&_{\text{pre}}\ (\texttt{lift}_{\text{pre}}\ f_2)$$

- Distributive properties makes proving this very easy:

  Theorem (correctness). For all e, $[\![e]\!]_{\text{pre}} = \texttt{lift}_{\text{pre}}\ [\![e]\!]$.

  In English: $[\![e]\!]_{\text{pre}}$ computes preimages under $[\![e]\!]$.

Correctness Theorems For Low, Low Prices

- Define $\texttt{lift}_{\text{pre}}\ f = f^{-1}$
- Derive $(\&\&\&_{\text{pre}})$ and others so that $\texttt{lift}_{\text{pre}}$ distributes; e.g.

$$\texttt{lift}_{\text{pre}}\ (f_1\ \&\&\&\ f_2)\ =\ (\texttt{lift}_{\text{pre}}\ f_1)\ \&\&\&_{\text{pre}}\ (\texttt{lift}_{\text{pre}}\ f_2)$$

- Distributive properties makes proving this very easy:

  Theorem (correctness). For all e, $[\![e]\!]_{\text{pre}} = \texttt{lift}_{\text{pre}}\ [\![e]\!]$.

  In English: $[\![e]\!]_{\text{pre}}$ computes preimages under $[\![e]\!]$.

- Other correctness proofs are similarly easy: prove 5 distributive properties

Correctness Theorems For Low, Low Prices

- Define $\texttt{lift}_{\text{pre}}\ f = f^{-1}$
- Derive $(\&\&\&_{\text{pre}})$ and others so that $\texttt{lift}_{\text{pre}}$ distributes; e.g.

$$\texttt{lift}_{\text{pre}}\ (f_1\ \&\&\&\ f_2) = (\texttt{lift}_{\text{pre}}\ f_1)\ \&\&\&_{\text{pre}}\ (\texttt{lift}_{\text{pre}}\ f_2)$$

- Distributive properties makes proving this very easy:

  Theorem (correctness). For all e, $[\![e]\!]_{\text{pre}} = \texttt{lift}_{\text{pre}}\ [\![e]\!]$.

  In English: $[\![e]\!]_{\text{pre}}$ computes preimages under $[\![e]\!]$.

- Other correctness proofs are similarly easy: prove 5 distributive properties
- Can add **`(random)`** and recursion to all semantics in one shot

Abstraction

Rectangle: An interval or union of intervals, a subset of `Bool`, or $A \times B$ for rectangles A and B

Abstraction

Rectangle: An interval or union of intervals, a subset of `Bool`, or $A \times B$ for rectangles A and B

- Easy representation; easy intersection, join (which overapproximates union), empty test, etc.

Abstraction

Rectangle: An interval or union of intervals, a subset of `Bool`, or $A \times B$ for rectangles A and B

- Easy representation; easy intersection, join (which overapproximates union), empty test, etc.
- Define $(\rightsquigarrow_{\widehat{\text{pre}}})$ (and therefore $[\![\cdot]\!]_{\widehat{\text{pre}}}$) by replacing sets and set operations with rectangles and rectangle operations

Abstraction

Rectangle: An interval or union of intervals, a subset of `Bool`, or $A \times B$ for rectangles A and B

- Easy representation; easy intersection, join (which overapproximates union), empty test, etc.
- Define $(\rightsquigarrow_{\widehat{\text{pre}}})$ (and therefore $[\![\cdot]\!]_{\widehat{\text{pre}}}$) by replacing sets and set operations with rectangles and rectangle operations
- Recursion is somewhat tricky—requires fine control over recursion depth or **`if`** choices

In Theory...

Theorem (sound). $[\![\cdot]\!]_{\widehat{\mathrm{pre}}}$ computes overapproximations of the preimages computed by $[\![\cdot]\!]_{\mathrm{pre}}$.

- Consequence: Sampling in abstract preimages doesn't leave anything out

In Theory...

Theorem (sound). $[\![\cdot]\!]_{\widehat{\text{pre}}}$ computes overapproximations of the preimages computed by $[\![\cdot]\!]_{\text{pre}}$.

- Consequence: Sampling in abstract preimages doesn't leave anything out

Theorem (decreasing). $[\![\cdot]\!]_{\widehat{\text{pre}}}$ never returns preimages larger than the given subdomain.

- Consequence: Refining abstract preimage sets never results in a worse approximation

In Theory...

Theorem (sound). $[\![\cdot]\!]_{\widehat{\mathrm{pre}}}$ computes overapproximations of the preimages computed by $[\![\cdot]\!]_{\mathrm{pre}}$.

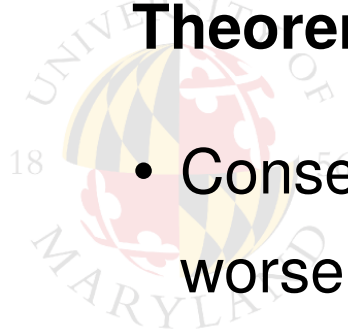
- Consequence: Sampling in abstract preimages doesn't leave anything out

Theorem (decreasing). $[\![\cdot]\!]_{\widehat{\mathrm{pre}}}$ never returns preimages larger than the given subdomain.

- Consequence: Refining abstract preimage sets never results in a worse approximation

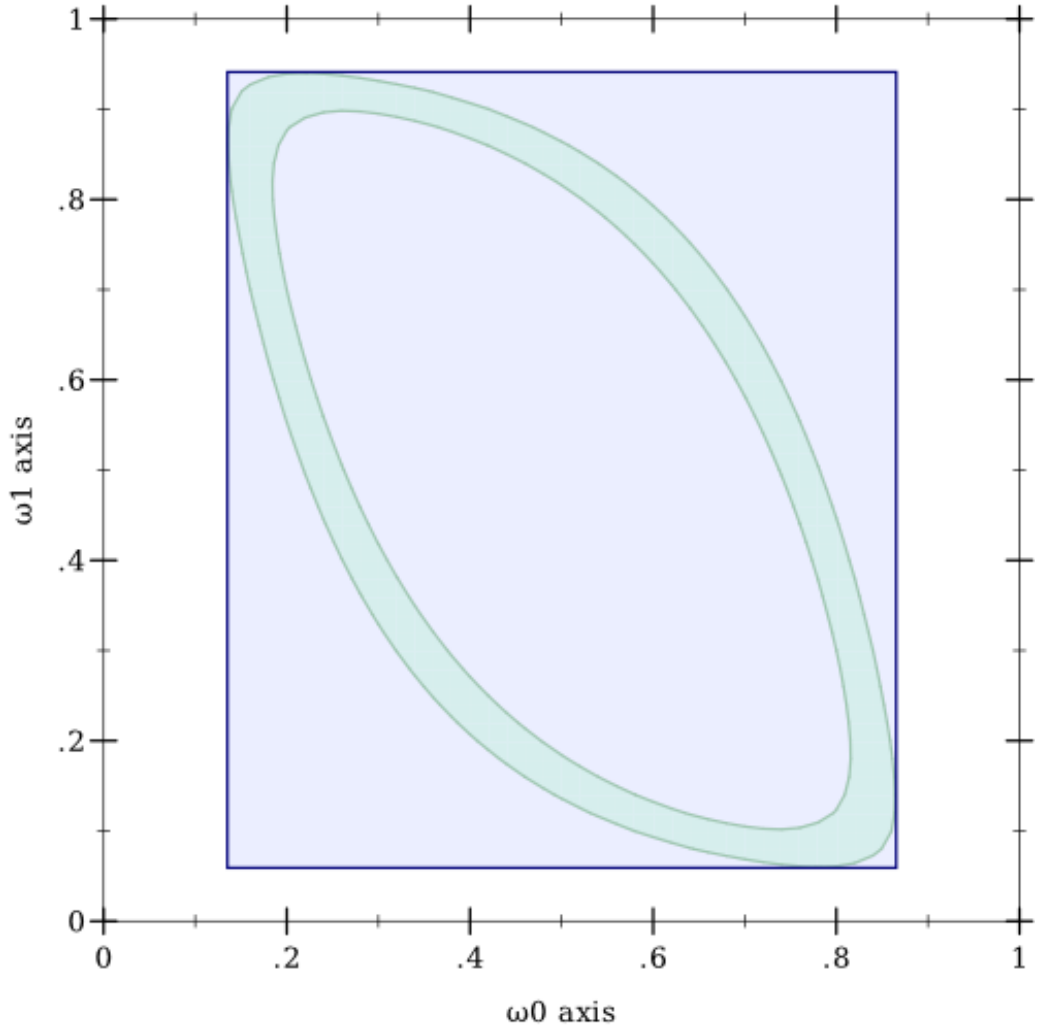
Theorem (monotone). $[\![\cdot]\!]_{\widehat{\mathrm{pre}}}$ is monotone.

- Consequence: *Partitioning* and then refining never results in a worse approximation



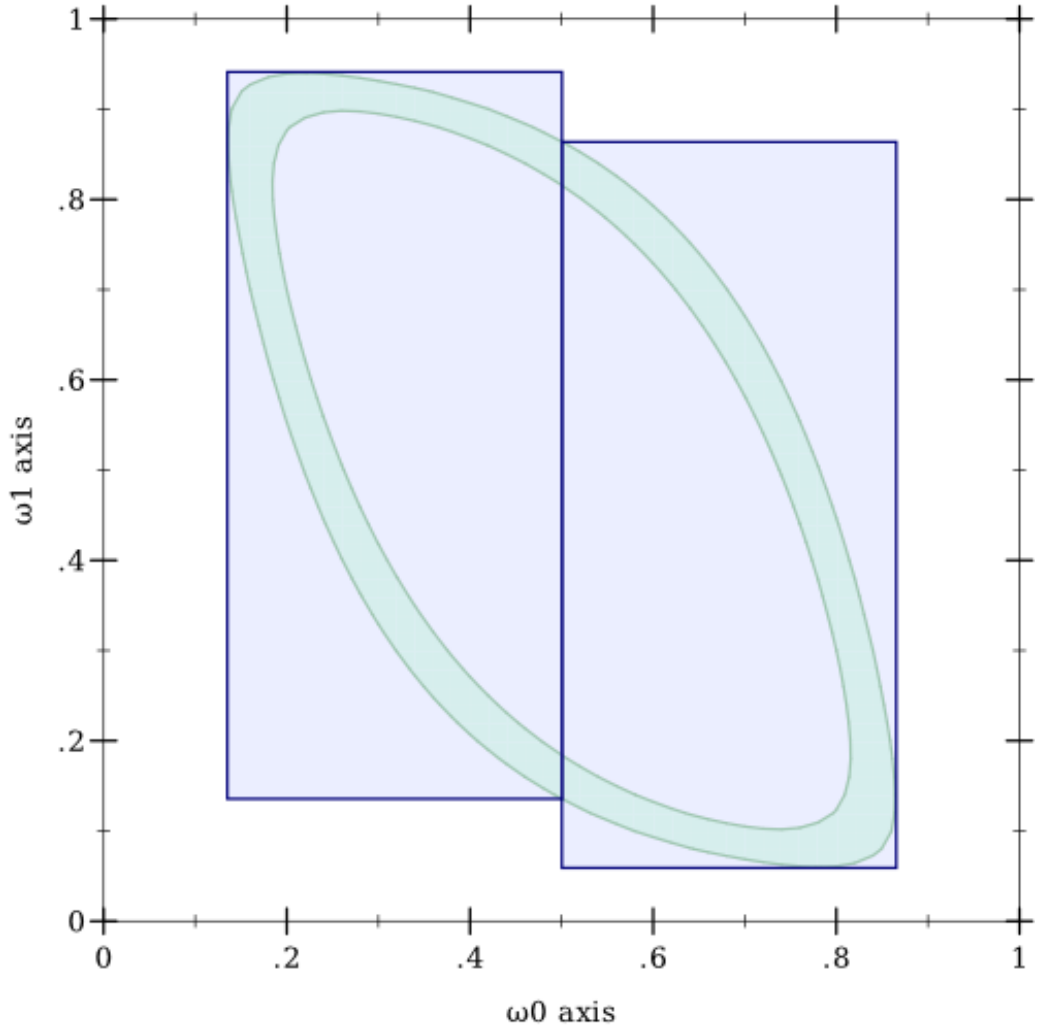
In Practice...

Theorems prove this always works:



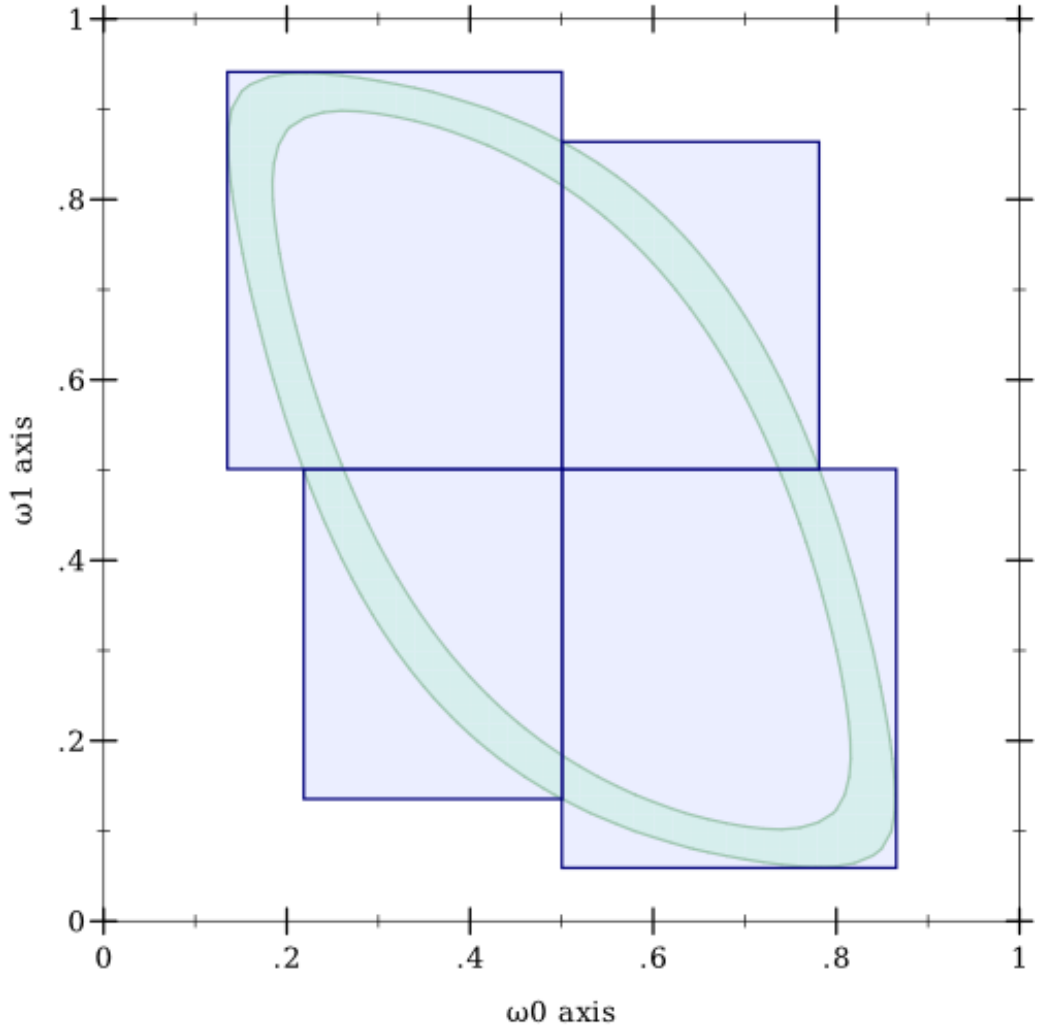
In Practice...

Theorems prove this always works:



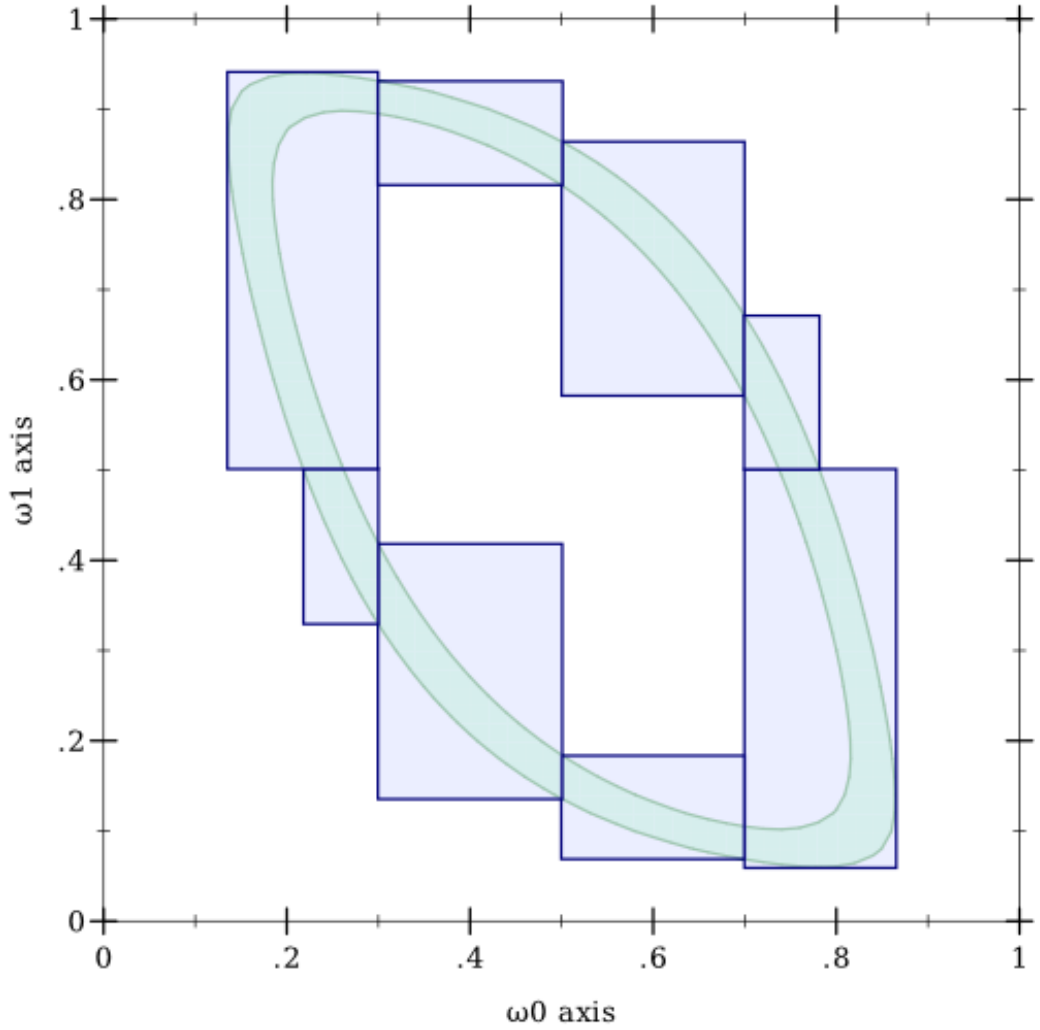
In Practice...

Theorems prove this always works:



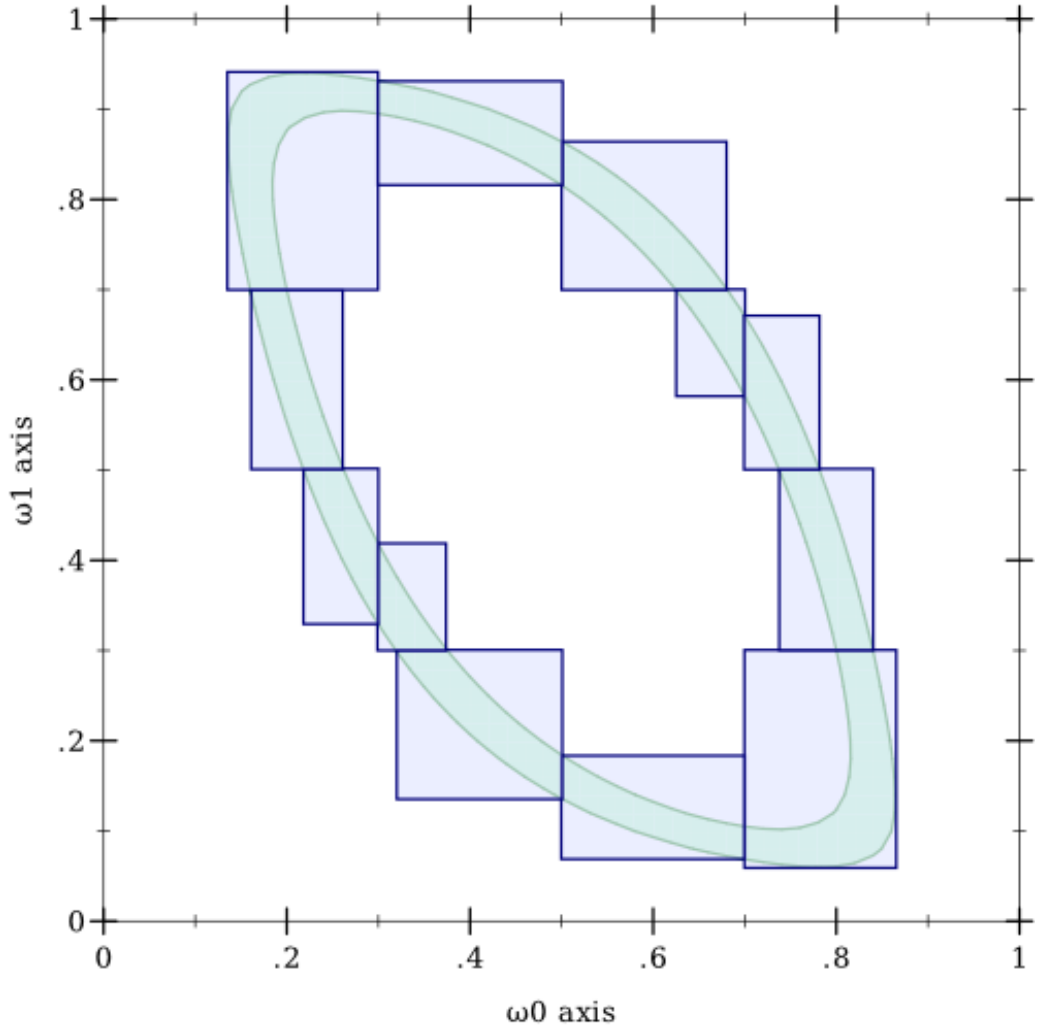
In Practice...

Theorems prove this always works:



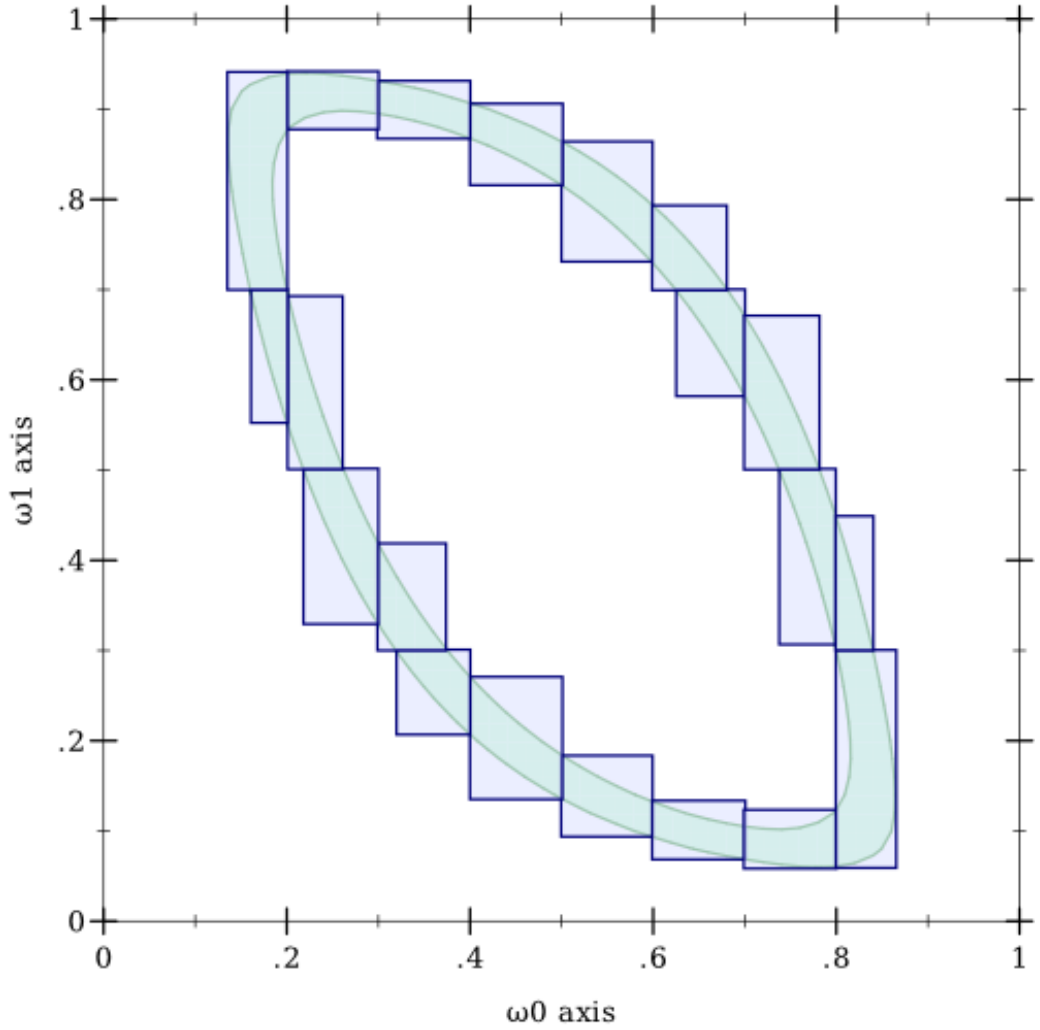
In Practice...

Theorems prove this always works:



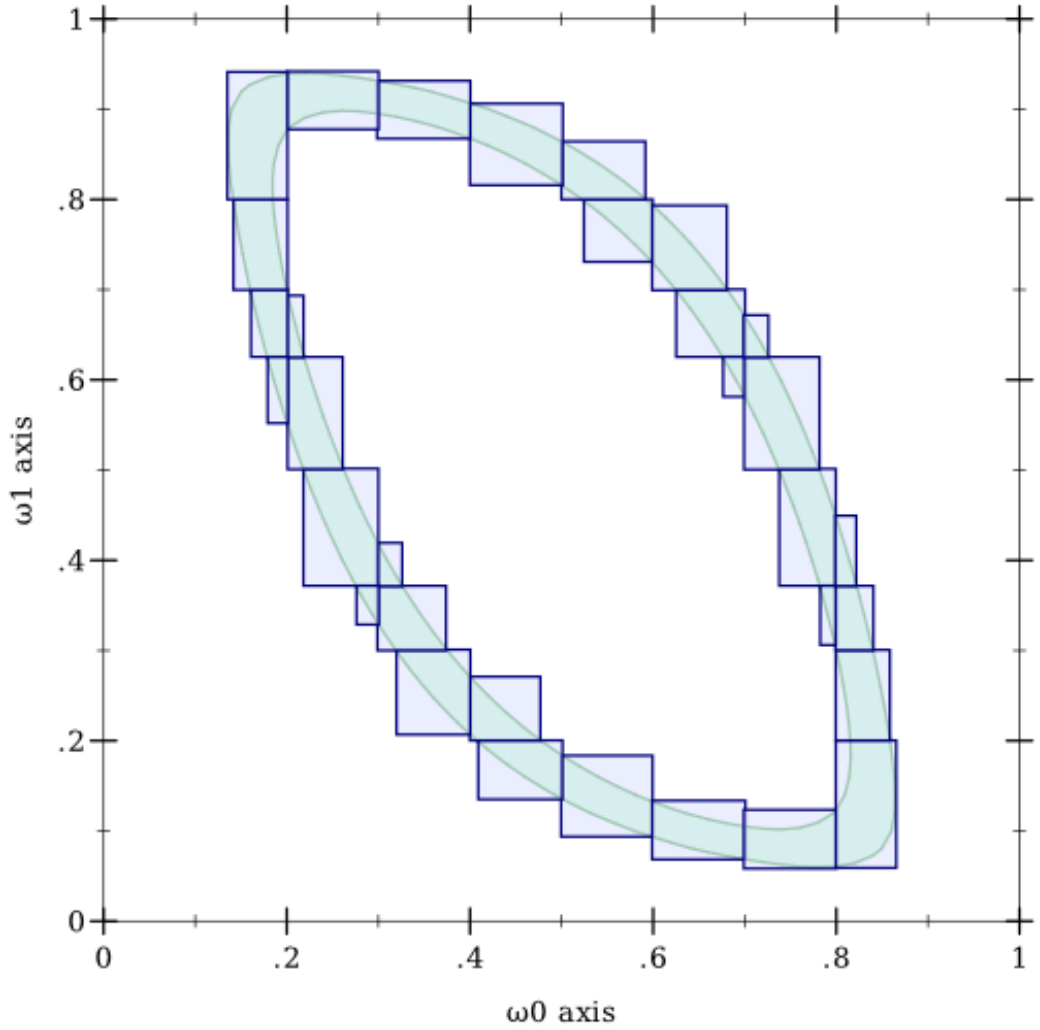
In Practice...

Theorems prove this always works:



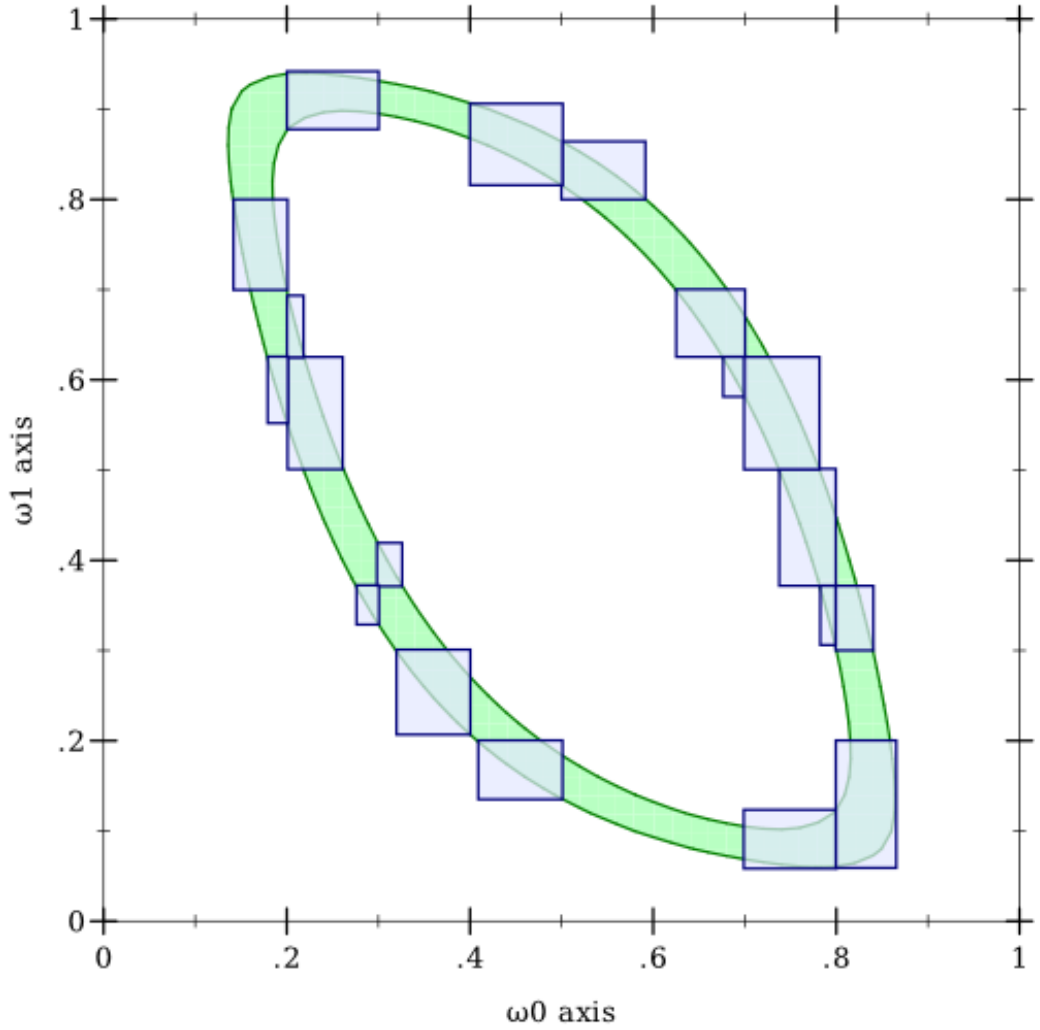
In Practice...

Theorems prove this always works:



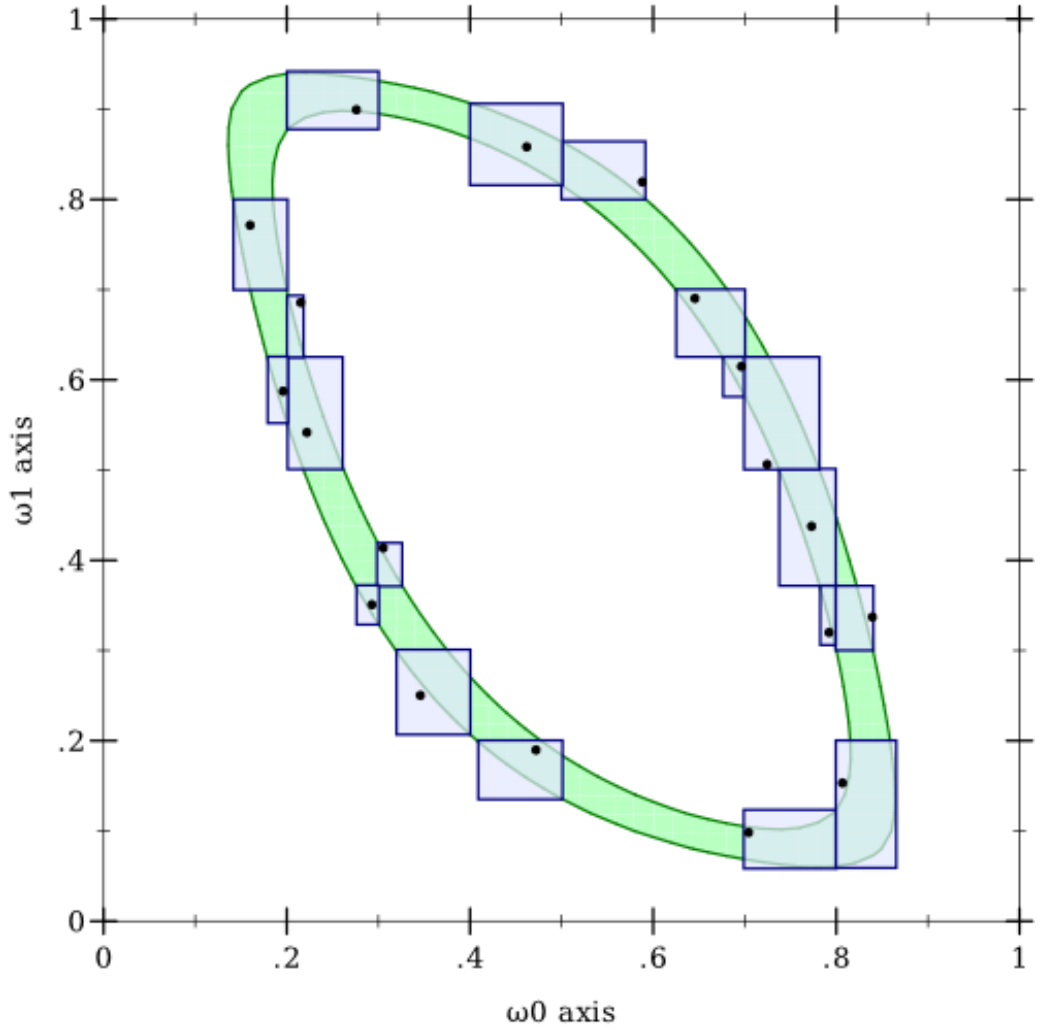
In Practice...

Theorems prove this always works:



In Practice...

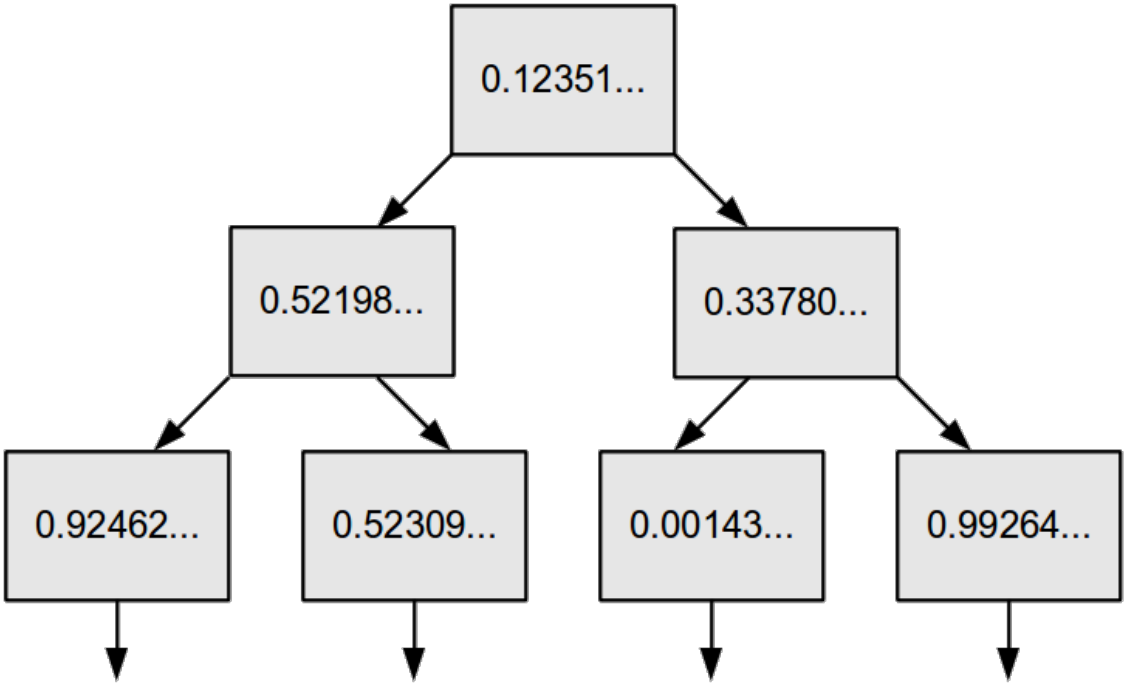
Theorems prove this always works:



Program Domain Values

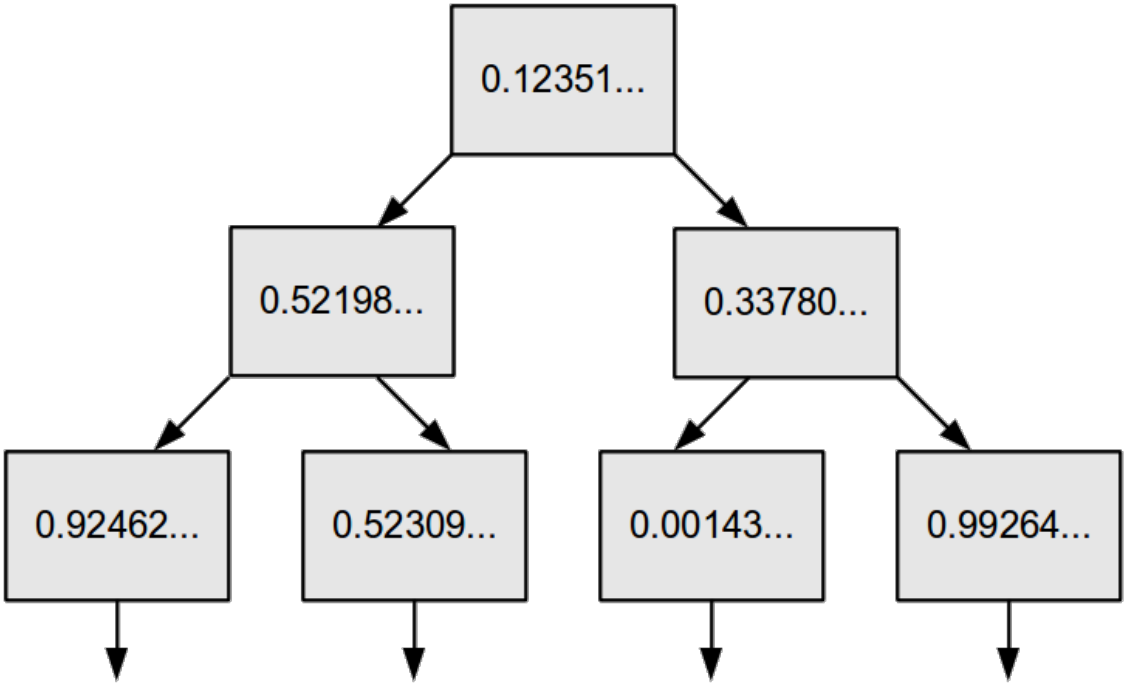
Program Domain Values

- Program inputs r are infinite binary trees:



Program Domain Values

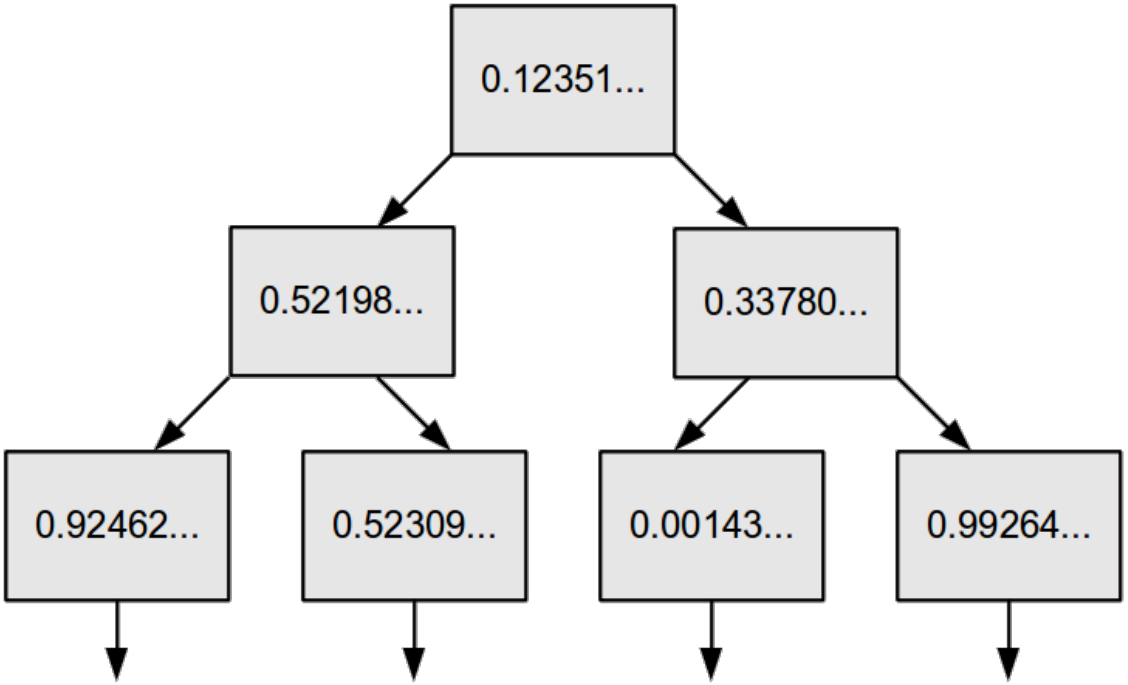
- Program inputs r are infinite binary trees:



- Every expression in a program is assigned a node

Program Domain Values

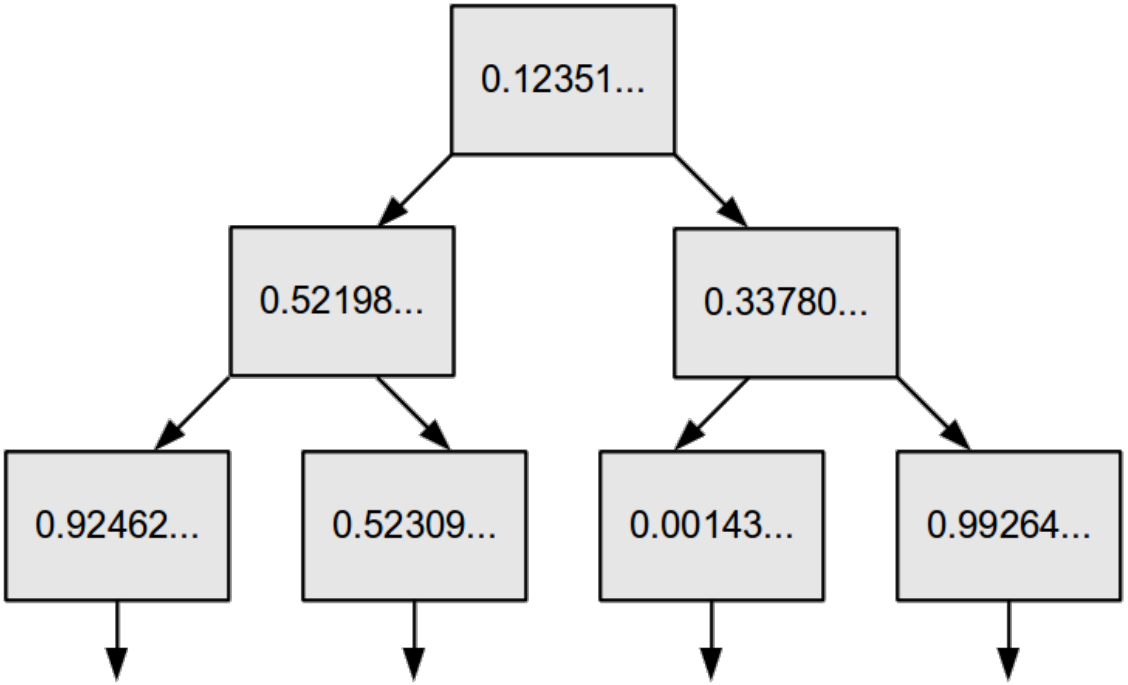
- Program inputs r are infinite binary trees:



- Every expression in a program is assigned a node
- Implemented using lazy trees of random values

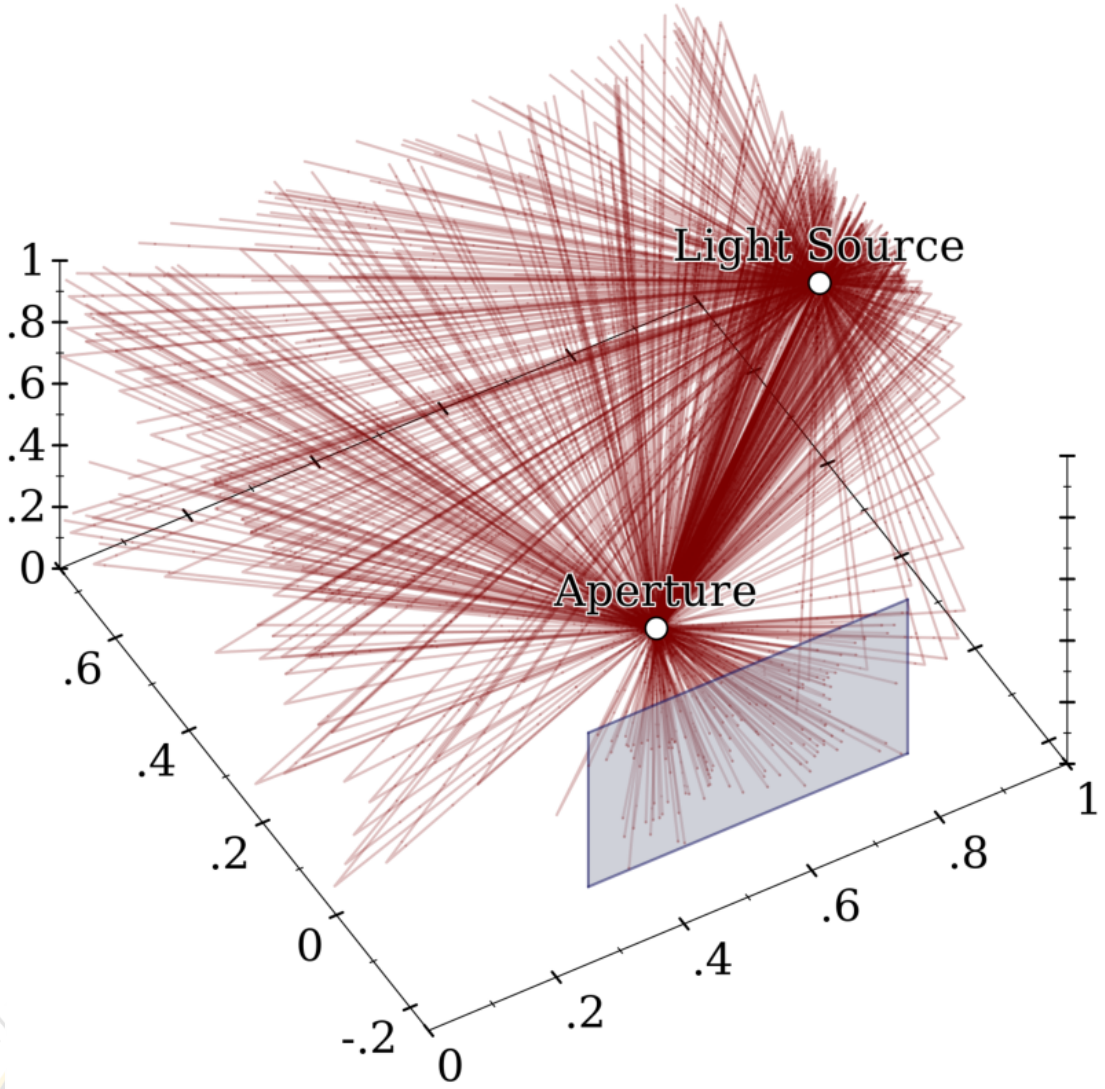
Program Domain Values

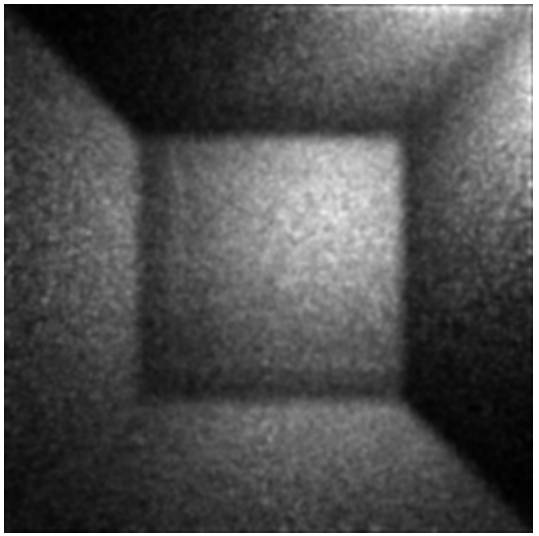
- Program inputs r are infinite binary trees:



- Every expression in a program is assigned a node
- Implemented using lazy trees of random values
- No probability *density* for domain, but there is a *measure*

Example: Stochastic Ray Tracing





Example: Probabilistic Verification

```
(struct/drbayes float-any ())
(struct/drbayes float (value error))
```


Example: Probabilistic Verification

```
(struct/drbayes float-any ())
(struct/drbayes float (value error))

(define/drbayes (flsqrt x)
  (if (float-any? x)
      x
      (let ([v  (float-value x)]
            [e  (float-error x)])
        (cond [(negative? v)  (float-any)]
              [(zero? v)      (float 0 0)]
              [else  (float (sqrt v)
                            (+ (- 1 (sqrt (- 1 e)))
                               (* 1/2 epsilon)))])))))
```


Example: Probabilistic Verification

```
(struct/drbayes float-any ())
(struct/drbayes float (value error))

(define/drbayes (flsqrt x)
  (if (float-any? x)
      x
      (let ([v  (float-value x)]
            [e  (float-error x)])
        (cond [(negative? v)  (float-any)]
              [(zero? v)      (float 0 0)]
              [else  (float (sqrt v)
                            (+ (- 1 (sqrt (- 1 e)))
                               (* 1/2 epsilon)))])))))
```

- Idea: sample `e` where `(> (float-error e) threshold)`

Example: Probabilistic Verification

```
(struct/drbayes float-any ())
(struct/drbayes float (value error))

(define/drbayes (flsqrt x)
  (if (float-any? x)
      x
      (let ([v  (float-value x)]
            [e  (float-error x)])
        (cond [(negative? v)  (float-any)]
              [(zero? v)      (float 0 0)]
              [else  (float (sqrt v)
                            (+ (- 1 (sqrt (- 1 e)))
                               (* 1/2 epsilon)))])))))
```

- Idea: sample `e` where `(> (float-error e) threshold)`
- Verified `flhypot`, `flsqrt1pm1`, `flsinh` in Racket's `math` library, as well as others

Examples: Other Inference Tasks

- Typical Bayesian inference
  - Hierarchical models
  - Bayesian regression
  - Model selection

Examples: Other Inference Tasks

- Typical Bayesian inference
  - Hierarchical models
  - Bayesian regression
  - Model selection
- Atypical
  - Programs that halt with probability < 1, or never halt
  - Probabilistic context-free grammars with context-sensitive constraints

Summary

- Probabilistic inference is hard, so PPLs have been popping up

Summary

- Probabilistic inference is hard, so PPLs have been popping up
- Interpreting every program requires measure theory

Summary

- Probabilistic inference is hard, so PPLs have been popping up
- Interpreting every program requires measure theory
- Defined a semantics that computes preimages

Summary

- Probabilistic inference is hard, so PPLs have been popping up
- Interpreting every program requires measure theory
- Defined a semantics that computes preimages
- Measuring abstract preimages or sampling in them carries out inference

Summary

- Probabilistic inference is hard, so PPLs have been popping up
- Interpreting every program requires measure theory
- Defined a semantics that computes preimages
- Measuring abstract preimages or sampling in them carries out inference
- Can do a lot of cool stuff that's normally inaccessible

https://github.com/ntoronto/drbayes